



DesignWare DW_apb_ssi Databook

DW_apb_ssi

Copyright Notice and Proprietary Information

Copyright © 2010 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.

Registered Trademarks (®)

Synopsys, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, CRITIC, Certify, CHIPit, Design Compiler, DesignWare, Formality, HDL Analyst, HSIM, HSPICE, Identify, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Syndicated, Synplicity, the Synplicity Logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Confirma, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclipse, Encore, EPIC, Galaxy, Galaxy Custom Designer, HANEX, HAPS, HapsTrak, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping Ssystem, HSIMplus, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, StarRC, Taurus, TotalRecall, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

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

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

PCI Express is a trademark of PCI-SIG.

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
Chapter 1	
Product Overview	11
1.1 DesignWare System Overview	11
1.2 General Product Description	13
1.2.1 DW_apb_ssi Block Diagram	13
1.3 Features	14
1.4 Standards Compliance	15
1.5 Verification Environment Overview	15
1.6 Licenses	15
1.7 Where To Go From Here	15
Chapter 2	
Building and Verifying a Component or Subsystem	17
2.1 Setting up Your Environment	17
2.2 Overview of the coreConsultant Configuration and Integration Process	18
2.2.1 coreConsultant Usage	18
2.2.2 Configuring the DW_apb_ssi within coreConsultant	19
2.2.3 Creating Gate-Level Netlists within coreConsultant	19
2.2.4 Verifying the DW_apb_ssi within coreConsultant	20
2.2.5 Running Leda on Generated Code with coreConsultant	20
2.3 Overview of the coreAssembler Configuration and Integration Process	21
2.3.1 coreAssembler Usage	21
2.3.2 Configuring the DW_apb_ssi within a Subsystem	24
2.3.3 Creating Gate-Level Netlists within coreAssembler	24
2.3.4 Verifying the DW_apb_ssi within coreAssembler	24
2.3.5 Running Leda on Generated Code with coreAssembler	25
2.4 Database Files	25
2.4.1 Design/HDL Files	25
2.4.2 Synthesis Files	26
2.4.3 Verification Reference Files	27
Chapter 3	
Functional Description	29
3.1 DW_apb_ssi Overview	29
3.1.1 Clock Ratios	31
3.1.2 Transmit and Receive FIFO Buffers	32
3.1.3 SSI Interrupts	33

3.2	Transfer Modes	34
3.2.1	Transmit and Receive	34
3.2.2	Transmit Only	34
3.2.3	Receive Only	35
3.2.4	EEPROM Read	35
3.3	Operation Modes	36
3.3.1	Serial-Master Mode	36
3.3.2	Serial-Slave Mode	46
3.4	Partner Connection Interfaces	49
3.4.1	Motorola Serial Peripheral Interface (SPI)	49
3.4.2	Texas Instruments Synchronous Serial Protocol (SSP)	54
3.4.3	National Semiconductor Microwire	55
3.5	DMA Controller Interface	65
3.5.1	Overview of Operation	65
3.5.2	Transmit Watermark Level and Transmit FIFO Underflow	67
3.5.3	Choosing the Transmit Watermark Level	67
3.5.4	Selecting DEST_MSIZ and Transmit FIFO Overflow	69
3.5.5	Receive Watermark Level and Receive FIFO Overflow	69
3.5.6	Choosing the Receive Watermark level	70
3.5.7	Selecting SRC_MSIZ and Receive FIFO Underflow	70
3.5.8	Handshaking Interface Operation	70
3.6	APB Interface	73
3.6.1	Control and Status Register APB Access	73
3.6.2	Data Register APB Access	74
Chapter 4		
Parameters		75
4.1	Parameter Descriptions	75
Chapter 5		
Signals		81
5.1	DW_apb_ssi Interface Diagram	81
5.2	DW_apb_ssi Signal Descriptions	83
Chapter 6		
Registers		91
6.1	Register Memory Map	91
6.2	Register and Field Descriptions	94
6.2.1	CTRLR0	94
6.2.2	CTRLR1	98
6.2.3	SSIENR	99
6.2.4	MWCR	100
6.2.5	SER	101
6.2.6	BAUDR	102
6.2.7	TXFTLR	103
6.2.8	RXFTLR	104
6.2.9	TXFLR	105
6.2.10	RXFLR	105
6.2.11	SR	106
6.2.12	IMR	108

6.2.13	ISR	109
6.2.14	RISR	110
6.2.15	TXOICR	111
6.2.16	RXOICR	112
6.2.17	RXUICR	112
6.2.18	MSTICR	113
6.2.19	ICR	113
6.2.20	DMACR	113
6.2.21	DMATDLR	114
6.2.22	DMARDLR	115
6.2.23	IDR	116
6.2.24	SSI_COMP_VERSION	117
6.2.25	DR	117
6.2.26	RX_SAMPLE_DLY	118
6.2.27	RSVD_0	119
6.2.28	RSVD_1	119
6.2.29	RSVD_2	120
Chapter 7		
Programming the DW_apb_ssi		121
7.1	Programming Considerations	121
Chapter 8		
Verification		123
8.1	Overview of Vera Tests	123
8.1.1	APB Interface	123
8.1.2	DW_apb_ssi as Master	124
8.1.3	DW_apb_ssi as Slave	124
8.1.4	DW_apb_ssi with DMA Interface	124
8.1.5	Interrupts	125
8.2	Overview of DW_apb_ssi Testbench	125
Chapter 9		
Integration Considerations		127
9.1	Reading and Writing from an APB Slave	127
9.1.1	Reading From Unused Locations	127
9.1.2	32-bit Bus System	128
9.1.3	16-bit Bus System	128
9.1.4	8-bit Bus System	129
9.2	Write Timing Operation	129
9.3	Read Timing Operation	130
9.4	Accessing Top-level Constraints	131
9.5	Coherency	132
9.5.1	Writing Coherently	132
9.5.2	Reading Coherently	138
Appendix A		
Application Notes		143
A.1	Interfacing DW_apb_ssi and Atmel SPI Devices	143
A.1.1	Synopsys SPI Operation	143
A.1.2	Atmel SPI Operation	144

A.1.3 Interoperability between DW_apb_ssi and Atmel Devices145

Appendix B

Glossary147

Index151

Preface

This databook provides information that you need to interface the DesignWare Synchronous Serial Interface (SSI), referred to as DW_apb_ssi throughout the remainder of this databook. This component conforms to the [AMBA Specification, Revision 2.0](#) from ARM.

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

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_ssi within the coreAssembler and coreConsultant tools.
- ❖ Chapter 3, “[Functional Description](#),” describes the functional operation of the DW_apb_ssi.
- ❖ Chapter 4, “[Parameters](#),” identifies the configurable parameters supported by the DW_apb_ssi.
- ❖ Chapter 5, “[Signals](#),” provides a list and description of the DW_apb_ssi signals.
- ❖ Chapter 6, “[Registers](#),” describes the programmable registers of the DW_apb_ssi.
- ❖ Chapter 7, “[Programming the DW_apb_ssi](#),” provides information needed to program the configured DW_apb_ssi.
- ❖ Chapter 8, “[Verification](#),” provides information on verifying the configured DW_apb_ssi.
- ❖ Chapter 9, “[Integration Considerations](#),” includes information you need to integrate the configured DW_apb_ssi into your design.
- ❖ Appendix A, “[Application Notes](#),” provides getting started information that allows you to walk through the process of using the DW_apb_ssi with Synopsys coreConsultant tool.
- ❖ Appendix B, “[Glossary](#),” provides a glossary of general terms.

Related Documentation

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

Document Revision History

This section tracks the significant documentation changes that occur from release-to-release and during a release from version 3.11b onward.

Table 1-1 Databook Revision History

Version	Databook Date	Description
3.18a	Dec 2010	Version change for 2010.12a release.
3.17a	Nov 2010	Corrected DW_ahb_dmac response in “Receive Watermark Level and Receive FIFO Overflow” section
3.17a	Oct 2010	Changes to clarify RTL enhancement for logic added to master state machine to prevent txd output from toggling when only receiving data frame
3.16a	Sept 2010	Corrected names of include files and vcs command used for simulation
3.15a	Mar 2010	Added a programmable delay register used to sample incoming data.
3.13a	Dec 2009	Updated databook to new template for consistency with other IIP/VIP/PHY databooks.
3.12a	July 2009	Corrected equations for avoiding underflow when programming a source burst transaction.
3.12a	May 2009	Removed references to QuickStarts, as they are no longer supported.
3.12a	Apr 2009	Enhanced overview and Figure 5-B.
3.12a	Oct 2008	Changed DR address offset from 0x60-0x15c to 0x60-0xfc; version change for 2008.10a release.
3.11c	June 2008	Version change for 2008.06a release.
3.11b	Mar 2008	DR register offset changed to 0x60 - 0x15c; occupies sixteen 32-bit addresses.
3.11b	Jan 2008	Updated to revised installation guide and consolidated release notes; changed references of “Designware AMBA” to simply “DesignWare.”
3.11b	June 2007	Version change for 2007.06a release.

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_ssi
 - ❖ **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>

1

Product Overview

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

1.1 DesignWare System Overview

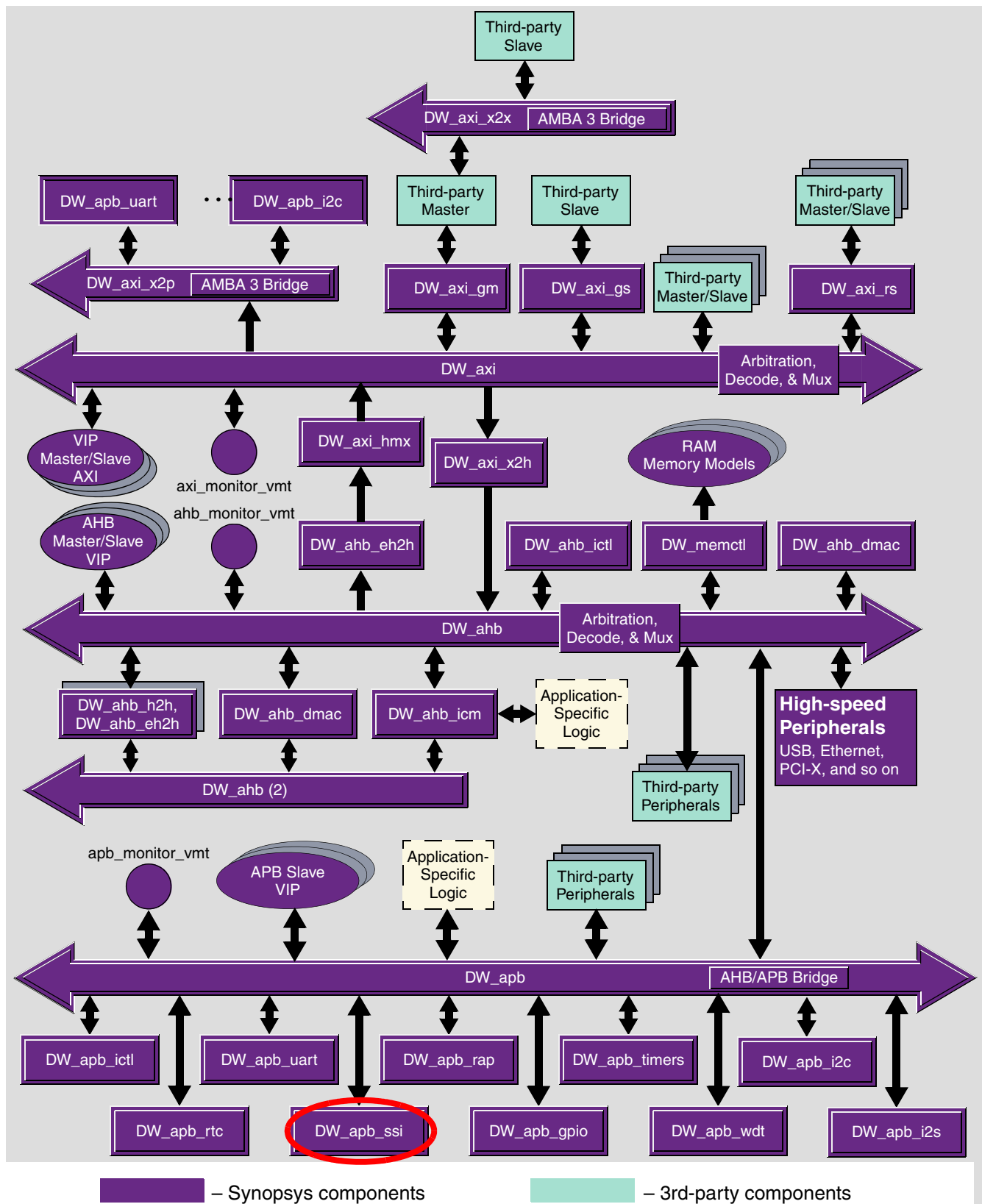
The Synopsys DesignWare Synthesizable Components environment is a parameterizable bus system containing AMBA version 2.0-compliant AHB (Advanced High-performance Bus) and APB (Advanced Peripheral Bus) components, and AMBA version 3.0-compliant AXI (Advanced eXtensible Interface) components.

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

**Attention**

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

Figure 1-1 Example of DW_apb_ssi in a Complete System



You can connect, configure, synthesize, and verify the DW_apb_ssi 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_ssi component, you might prefer to use coreConsultant, documentation for which is available in the [coreConsultant User Guide](#).

1.2 General Product Description

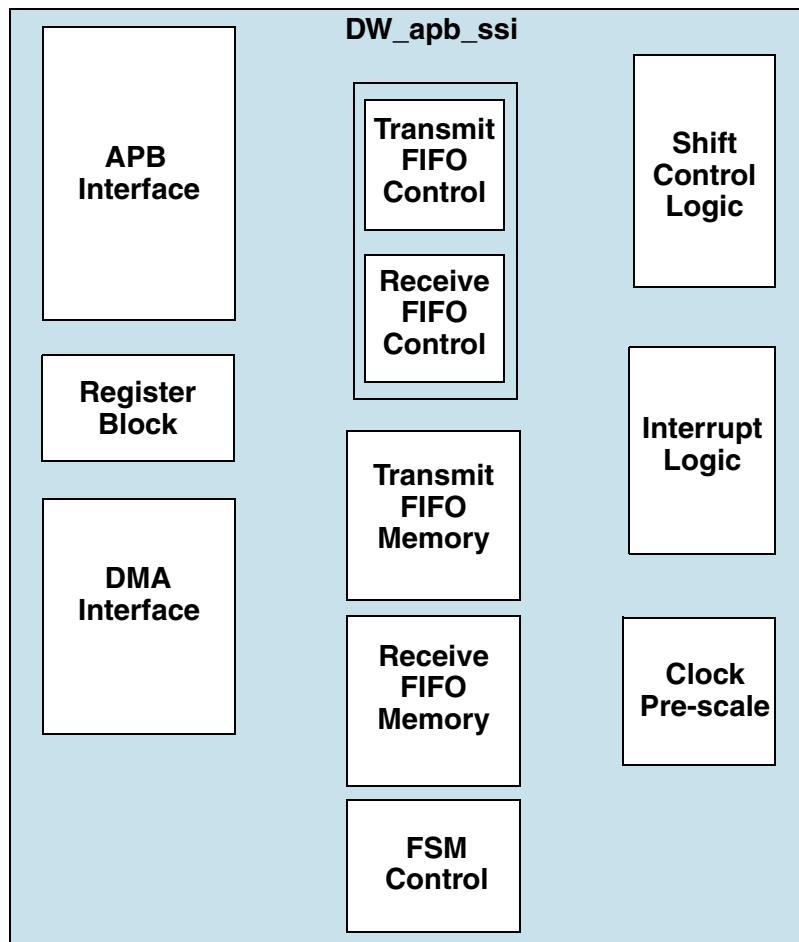
The Synopsys DW_apb_ssi is a component of the DesignWare Advanced Peripheral Bus (DW_apb) and conforms to the [AMBA Specification, Revision 2.0](#) from ARM.

1.2.1 DW_apb_ssi Block Diagram

Figure 1-2 shows the following functional groupings of the main interfaces to the DW_apb_ssi block:

- ❖ APB interface and DMA Controller Interface
- ❖ Transmit and receive FIFO controllers and an FSM controller
- ❖ Register block
- ❖ Shift control and interrupt logic

Figure 1-2 DW_apb_ssi Block Diagram



1.3 Features

DW_apb_ssi has the following features:

- ❖ APB interface – Allows for easy integration into a DesignWare Synthesizable Components for AMBA 2 implementation.
- ❖ Scalable APB data bus width – Supports APB data bus widths of 8, 16, and 32 bits.
- ❖ Serial-master or serial-slave operation – Enables serial communication with serial-master or serial-slave peripheral devices.
- ❖ DMA Controller Interface – Enables the DW_apb_ssi to interface to a DMA controller over the bus using a handshaking interface for transfer requests.
- ❖ Independent masking of interrupts – Master collision, transmit FIFO overflow, transmit FIFO empty, receive FIFO full, receive FIFO underflow, and receive FIFO overflow interrupts can all be masked independently.
- ❖ Multi-master contention detection – Informs the processor of multiple serial-master accesses on the serial bus.
- ❖ Bypass of meta-stability flip-flops for synchronous clocks – When the APB clock (pclk) and the DW_apb_ssi serial clock (ssi_clk) are synchronous, meta-stable flip-flops are not used when transferring control signals across these clock domains.
- ❖ Programmable delay on the sample time of the received serial data bit (rxd), when configured in Master Mode; enables programmable control of routing delays resulting in higher serial data-bit rates.
- ❖ Programmable features:
 - ◆ Serial interface operation – Choice of Motorola SPI, Texas Instruments Synchronous Serial Protocol or National Semiconductor Microwire.
 - ◆ Clock bit-rate – Dynamic control of the serial bit rate of the data transfer; used in only serial-master mode of operation.
 - ◆ Data Item size (4 to 16 bits) – Item size of each data transfer under the control of the programmer.
- ❖ Configurable features:
 - ◆ FIFO depth – Configurable depth of the transmit and receive FIFO buffers from 2 to 256 words deep. The FIFO width is fixed at 16 bits.
 - ◆ Number of slave select outputs – When operating as a serial master, 1 to 16 serial slave-select output signals can be generated.
 - ◆ Hardware/software slave-select – Dedicated hardware slave-select lines can be used or software control can be used to target the serial-slave device.
 - ◆ Combined or individual interrupt lines – You may choose to bring all individual interrupt lines or one combined interrupt line from the DW_apb_ssi to the interrupt controller.
 - ◆ Interrupt polarity – This configuration option selects the serial-clock phase of the SPI format directly after reset.

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

1.4 Standards Compliance

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

1.5 Verification Environment Overview

The DW_apb_ssi 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 [123](#) section discusses the specific procedures for verifying the DW_apb_ssi.

1.6 Licenses

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

1.7 Where To Go From Here

At this point, you may want to get started working with the DW_apb_ssi component within a subsystem or by itself. Synopsys provides several tools within its coreTools suite of products for the purposes of configuration, synthesis, and verification of single or multiple synthesizable IP components – coreConsultant and coreAssembler. For information on the different coreTools, refer to [Guide to coreTools Documentation](#).

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

For more information about implementing your DW_apb_ssi component within a DesignWare subsystem using coreAssembler, refer to “[Overview of the coreAssembler Configuration and Integration Process](#)” on page [21](#).

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_ssi 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 SYNOPSYS. If you are not familiar with these requirements and the necessary licenses, refer to [“Setting up Your Environment”](#) in the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide*.

2.2 Overview of the coreConsultant Configuration and Integration Process

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

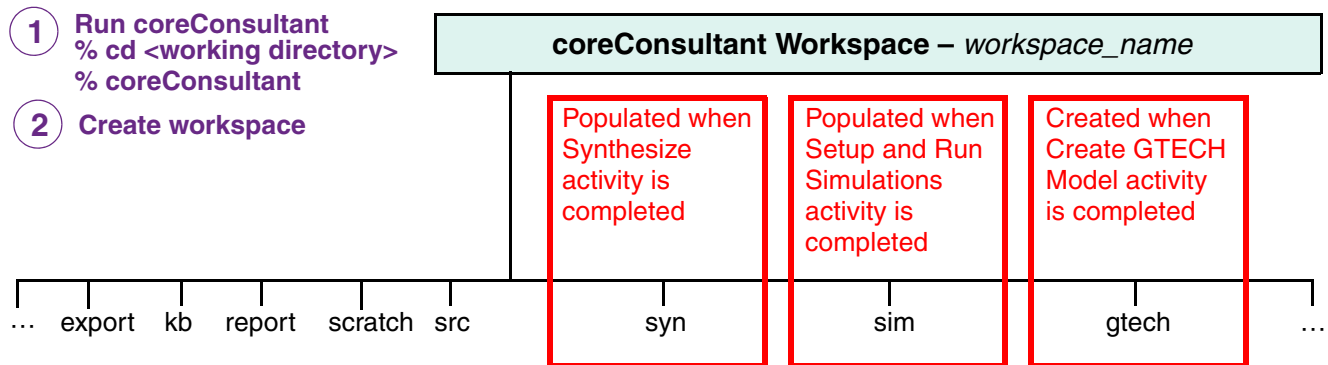


Table 2-1 provides a description of the implementation workspace directory and subdirectories.

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 25.

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

2.2.2 Configuring the DW_apb_ssi within coreConsultant

The “[Parameters](#)” chapter on [page 75](#) describes the DW_apb_ssi 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_ssi.

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_ssi.

2.2.4 Verifying the DW_apb_ssi within coreConsultant

The “[Verification](#)” chapter on [page 123](#) provides an overview of the testbench available for DW_apb_ssi 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_ssi.

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.

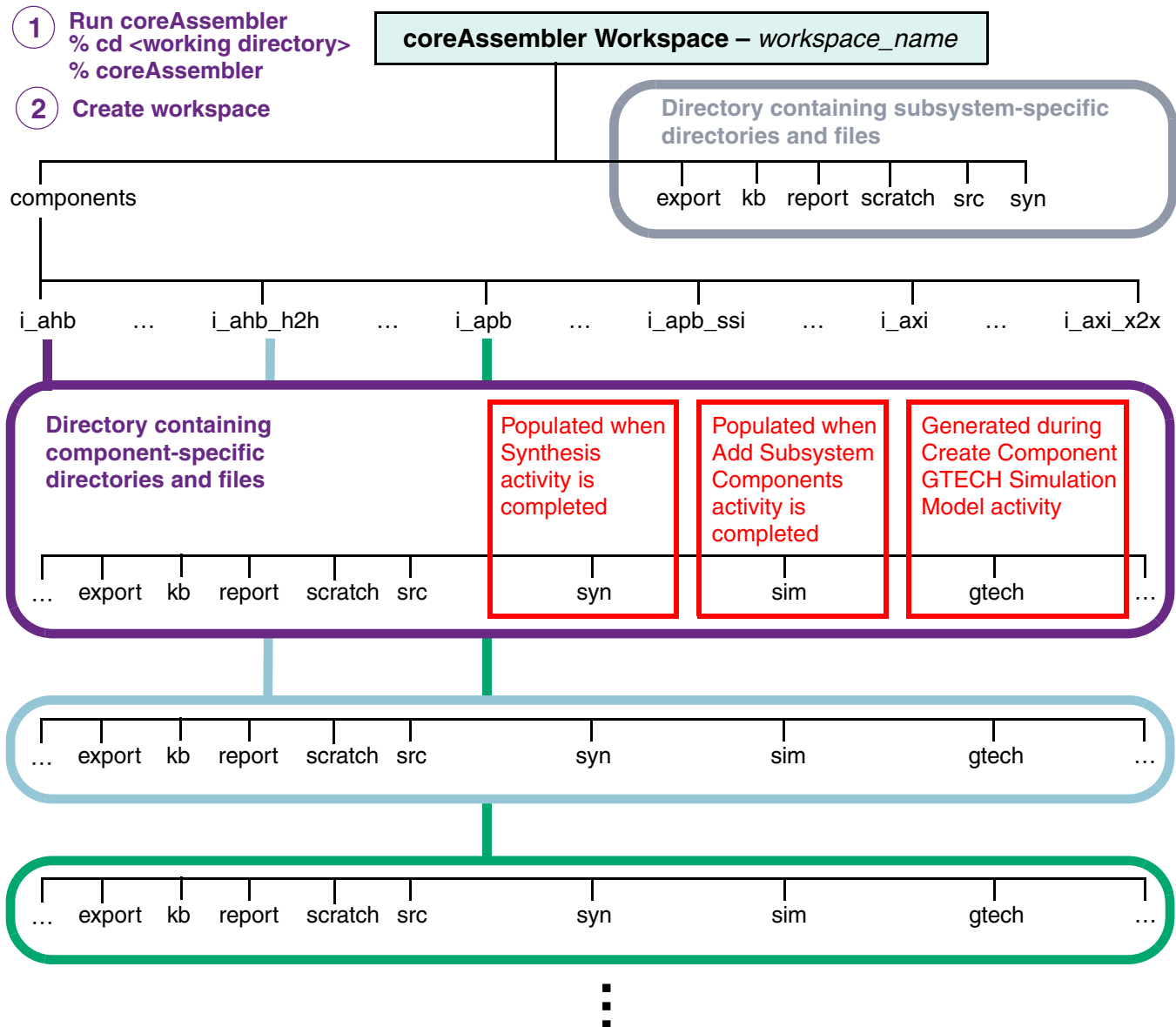
2.3 Overview of the coreAssembler Configuration and Integration Process

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



3 Use coreAssembler to create, synthesize, and verify your subsystem

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</i> /auxiliary	Scripts and text files used by coreAssembler. Generated during Add Subsystem Components activity.
<i>i_component</i> /doc	Contains local copies of component-specific databooks. Generated during Add Subsystem Components activity.
<i>i_component</i> /export	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</i> /gtech	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</i> /kb	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</i> /leda	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</i> /pkg	Contains RTL preprocessor scripts. Generated during Configure Components activity.
<i>i_component</i> /report	Contains all of the reports created by coreAssembler during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated during Add Subsystem Components activity; populated and updated throughout activities.
<i>i_component</i> /scratch	Contains temp files used during the coreAssembler processes. Generated during Add Subsystem Components activity; populated and updated throughout activities.
<i>i_component</i> /sim	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)

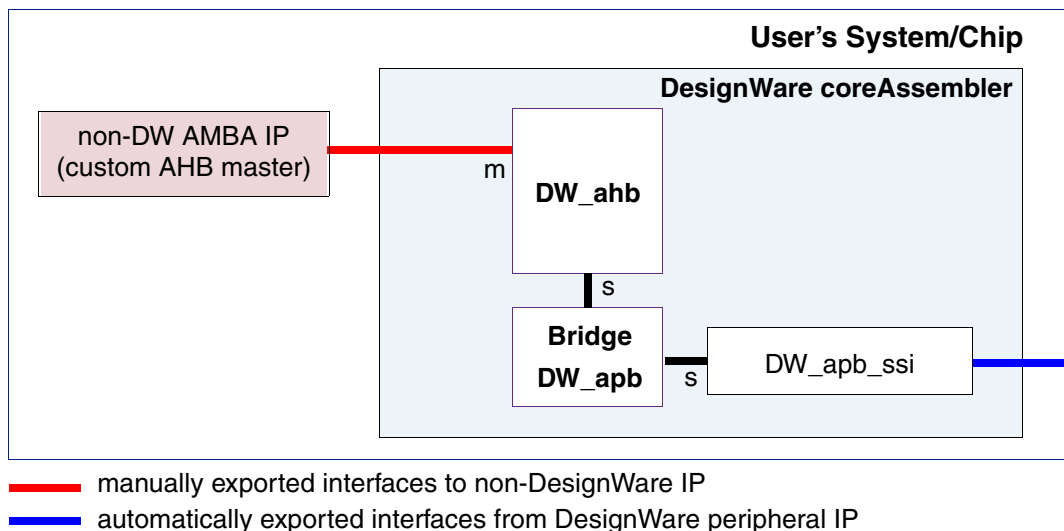
Directory/Subdirectory	Description
i_component/src	Includes the top-level RTL file, <i>design_name.v</i> . If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated during Add Subsystem Components activity; populated during Specify Configuration activity.
i_component/syn	Contains synthesis files for the component. Generated during Add Subsystem Components activity; updated during Synthesis activity.
i_component/tcl	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 25.

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_ssi in a simple subsystem.

Figure 2-3 DW_apb_ssi 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_ssi
- ❖ 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_ssi within a Subsystem

The “Parameters” chapter on page 75 describes the DW_apb_ssi 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_ssi within coreAssembler

The “Verification” chapter on page 123 provides an overview of the testbench available for DW_apb_ssi 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 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

Table 2-3 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. When you include the component in your simulation, you must 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
<i>./src/component_submodule.v</i>	Yes	Sub-modules of component
<i>./src/component_constants.v</i>	No	Includes the constants used internally in the design.
<i>./src/component_undef.v</i>		Includes an undef for each of the definitions found in the <i>component_cc_constants.v</i> file; compiled in after the last file listed in <i>./src/components.lst</i> when compiling multiple instances of the same IP.

Table 2-3 RTL-Level Files (Continued)

Files	Encrypted?	Purpose
<i>./src/component.lst</i>	No	Lists the order in which the RTL files should be read into tools, such as simulators or dc_shell. For example, use the following option to read the design into VCS: <code>vcs +v2k -f component.lst</code>

2.4.1.2 Simulation Model Files

Table 2-4 includes the simulation model 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
<i>./gtech/final/db/component.v</i>	No	Simulation model of the component for use with non-Synopsys simulators. A technology-independent, gate-level netlist. VHDL and Verilog versions are generated. When you use this simulation model in your simulation, you must include the DesignWare libraries by using the following options in your simulator invocation: <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

Table 2-5 includes the files that are generated after the Create Gate-Level Netlist activity is performed on a component.

Table 2-5 Synthesis Files

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

2.4.3 Verification Reference Files

The files described in [Table 2-6](#) 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 operation of the DW_apb_ssi.

The DW_apb_ssi is a configurable, synthesizable, and programmable component that is a full-duplex master or slave-synchronous serial interface. The host processor accesses data, control, and status information on the DW_apb_ssi through the APB interface. The DW_apb_ssi may also interface with a DMA Controller using an optional set of DMA signals, which can be selected at configuration time.

As described in more detail later, the DW_apb_ssi can be configured in one of two modes of operations: as a serial master or a serial slave. The DW_apb_ssi can connect to any serial-master or serial-slave peripheral device using one of the following interfaces:

- ❖ Motorola Serial Peripheral Interface (SPI)
- ❖ Texas Instruments Serial Protocol (SSP)
- ❖ National Semiconductor Microwire

3.1 DW_apb_ssi Overview

In order for the DW_apb_ssi to connect to a serial-master or serial-slave peripheral device, the peripheral must have a least one of the following interfaces:

- ❖ Motorola Serial Peripheral Interface (SPI) – A four-wire, full-duplex serial protocol from Motorola. There are four possible combinations for the serial clock phase and polarity. The clock phase (SCPH) determines whether the serial transfer begins with the falling edge of the slave select signal or the first edge of the serial clock. The slave select line is held high when the DW_apb_ssi is idle or disabled. For more information, refer to [“Motorola Serial Peripheral Interface \(SPI\)”](#) on page 49.
- ❖ Texas Instruments Serial Protocol (SSP) – A four-wire, full-duplex serial protocol. The slave select line used for SPI and Microwire protocols doubles as the frame indicator for the SSP protocol. For more information, refer to [“Texas Instruments Synchronous Serial Protocol \(SSP\)”](#) on page 54.
- ❖ National Semiconductor Microwire – A half-duplex serial protocol, which uses a control word transmitted from the serial master to the target serial slave. For more information, refer to [“National Semiconductor Microwire”](#) on page 55.

You can program the FRF (frame format) bit field in the Control Register 0 (CTRLR0) to select which protocol is used. You specify the FRF at configuration time to be hardcoded or programmable by setting the SSI_HC_FRF parameter. For more information about this configuration parameter, refer to [Table 4-1](#) on page 75.

The serial protocols supported by the DW_apb_ssi allow for serial slaves to be selected or addressed using either hardware or software. When implemented in hardware, serial slaves are selected under the control of dedicated hardware select lines. The number of select lines generated from the serial master is equal to the number of serial slaves present on the bus. The serial-master device asserts the select line of the target serial slave before data transfer begins. This architecture is illustrated in [Figure 3-1\(A\)](#) on [page 31](#).

When implemented in software, the input select line for all serial slave devices should originate from a single slave select output on the serial master. In this mode it is assumed that the serial master has only a single slave select output. If there are multiple serial masters in the system, the slave select output from all masters can be logically ANDed to generate a single slave select input for all serial slave devices.

The main program in the software domain controls selection of the target slave device; this architecture is illustrated in [Figure 3-1\(B\)](#) on [page 31](#). Software would use the SSIENR register in all slaves in order to control which slave is to respond to the serial transfer request from the master device.

The following example is pseudo code that illustrates how to use software to select the target slave.

3.1.0.1 Example of Target Slave Selection Using Software

```
int main() {

// This function sets the SSI_EN bit to logic '0' in the SSIENR register
// of each device on the serial bus

disable_all_serial_devices();

// This function initializes the master device for the serial transfer
// 1. Write CTRLR0 to match the required transfer
// 2. If transfer is receive only write number of frames into CTRLR1
// 3. Write BAUDR to set the transfer baud rate.
// 4. Write TXFTLR and RXFTLR to set FIFO threshold levels
// 5. Write IMR register to set interrupt masks
// 6. Write SER register bit[0] to logic '1'
// 7. Write SSIENR register bit[0] to logic '1' to enable the master.

initialize_mst(ssi_mst_1);

// This function initializes the target slave device (slave 1 in this example)
// for the serial transfer.
// 1. Write CTRLR0 to match the required transfer
// 2. Write TXFTLR and RXFTLR to set FIFO threshold levels
// 3. Write IMR register to set interrupt masks
// 4. Write SSIENR register bit[0] to logic '1' to enable the slave.
// 5. If the slave is to transmit data, write data into TX FIFO
// Now the slave is enabled and awaiting an active level on its
// ss_in_n input port. Note all other serial slaves are disabled (SSI_EN=0)
// and therefore will not respond to an active level on their ss_in_n port.

initialize_slv(ssi_slv_1);

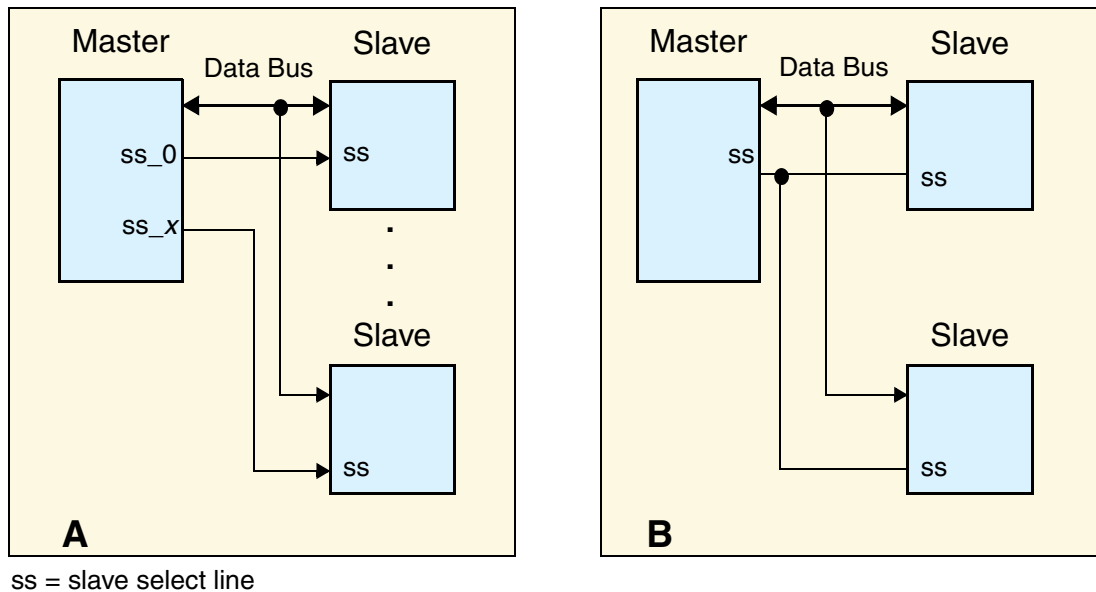
// This function begins the serial transfer by writing transmit data into
// the master's TX FIFO.

start_serial_xfer(ssi_mst_1);
```

```
// User can poll the busy status with a function or use an ISR to determine
// when the serial transfer has completed.
}
```

The DW_apb_ssi does not enforce hardware or software control for serial-slave device selection. You can configure the DW_apb_ssi for either implementation.

Figure 3-1 Hardware/Software Slave Selection



3.1.1 Clock Ratios

The frequency of the DW_apb_ssi serial input clock (ssi_clk) must be less than or equal to the frequency of pclk, which guarantees that control signals from the ssi_clk domain are synchronized to the pclk domain. When pclk and ssi_clk are asynchronous, synchronization logic transfers control signals from one clock domain to the other. If both clocks are synchronous, the synchronization logic is unneeded, thus reducing latency in the peripheral.

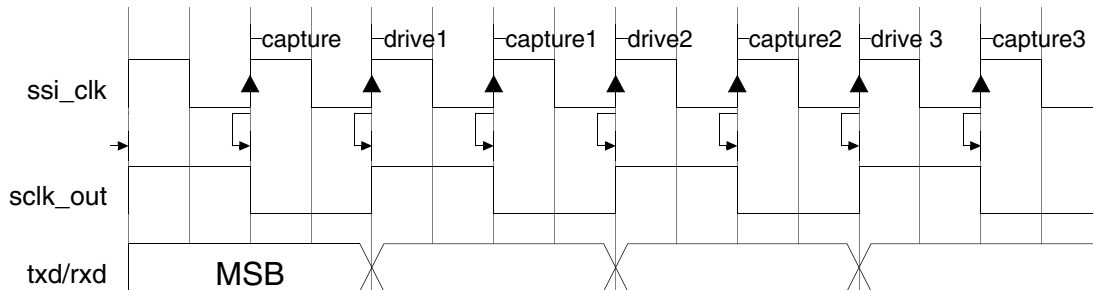
When DW_apb_ssi is configured as a master device, the maximum frequency of the bit-rate clock (sclk_out) is one-half the frequency of ssi_clk. This allows the shift control logic to capture data on one clock edge of sclk_out and propagate data on the opposite edge; this is illustrated in [Figure 3-2](#) on page 32. The sclk_out line toggles only when an active transfer is in progress. At all other times it is held in an inactive state, as defined by the serial protocol under which it operates.

The frequency of sclk_out can be derived from the following equation:

$$F_{\text{sclk_out}} = \frac{F_{\text{ssi_clk}}}{\text{SCKDV}}$$

SCKDV is a bit field in the programmable register [BAUDR](#), holding any even value in the range 0 to 65,534. If SCKDV is 0, then sclk_out is disabled.

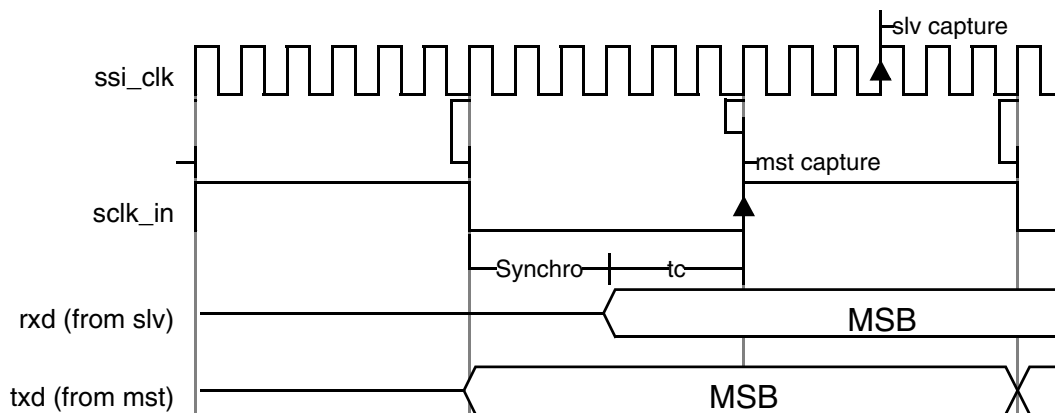
Figure 3-2 Maximum sclk_out/ssi_clk Ratio



When DW_apb_ssi is configured as a slave device, the minimum frequency of ssi_clk depends on the operation of the slave peripheral. If the slave device is *receive only*, the minimum frequency of ssi_clk is six times the maximum expected frequency of the bit-rate clock from the master device (sclk_in). The sclk_in signal is double synchronized to the ssi_clk domain, and then it is edge detected; this synchronization requires three ssi_clk periods.

If the slave device is *transmit and receive*, the minimum frequency of ssi_clk is eight times the maximum expected frequency of the bit-rate clock from the master device (sclk_in). This ensures that data on the master rxd line is stable before the master shift control logic captures the data. This is illustrated in [Figure 3-3](#).

Figure 3-3 Slave ssi_clk/sclk_in Ratio



A summary of the frequency ratio restrictions between the bit-rate clock (sclk_out/sclk_in) and the DW_apb_ssi peripheral clock (ssi_clk) are described as:

- ❖ Master: $F_{ssi_clk} \geq 2 \times (\text{maximum } F_{sclk_out})$
- ❖ Slave (receive only): $F_{ssi_clk} \geq 6 \times (\text{maximum } F_{sclk_in})$
- ❖ Slave: $F_{ssi_clk} \geq 8 \times (\text{maximum } F_{sclk_in})$

3.1.2 Transmit and Receive FIFO Buffers

The FIFO buffers used by the DW_apb_ssi are internal D-type flip-flops that can be configured in depth between 2 to 256. The width of both transmit and receive FIFO buffers is fixed at 16 bits due to the serial specifications, which state that a serial transfer (data frame) can be 4 to 16 bits in length. Data frames that are

less than 16 bits in size must be right-justified when written into the transmit FIFO buffer. The shift control logic automatically right-justifies receive data in the receive FIFO buffer.

Each data entry in the FIFO buffers contains a single data frame. It is impossible to store multiple data frames in a single FIFO location; for example, you may not store two 8-bit data frames in a single FIFO location. If an 8-bit data frame is required, the upper 8-bits of the FIFO entry are ignored or unused when the serial shifter transmits the data.

**Note**

The transmit and receive FIFO buffers are cleared when the DW_apb_ssi is disabled (SSI_EN = 0) or when it is reset (presetrn).

The transmit FIFO is loaded by APB write commands to the DW_apb_ssi data register (DR). Data are popped (removed) from the transmit FIFO by the shift control logic into the transmit shift register. The transmit FIFO generates a FIFO empty interrupt request (ssi_txe_intr) when the number of entries in the FIFO is less than or equal to the FIFO threshold value. The threshold value, set through the programmable register TXFTLR, determines the level of FIFO entries at which an interrupt is generated. The threshold value allows you to provide early indication to the processor that the transmit FIFO is nearly empty. A transmit FIFO overflow interrupt (ssi_txo_intr) is generated if you attempt to write data into an already full transmit FIFO.

Data are popped from the receive FIFO by APB read commands to the DW_apb_ssi data register (DR). The receive FIFO is loaded from the receive shift register by the shift control logic. The receive FIFO generates a FIFO-full interrupt request (ssi_rxf_intr) when the number of entries in the FIFO is greater than or equal to the FIFO threshold value plus 1. The threshold value, set through programmable register RXFTLR, determines the level of FIFO entries at which an interrupt is generated.

The threshold value allows you to provide early indication to the processor that the receive FIFO is nearly full. A receive FIFO overrun interrupt (ssi_rxo_intr) is generated when the receive shift logic attempts to load data into a completely full receive FIFO. However, this newly received data are lost. A receive FIFO underflow interrupt (ssi_rxu_intr) is generated if you attempt to read from an empty receive FIFO. This alerts the processor that the read data are invalid.

3.1.3 SSI Interrupts

The DW_apb_ssi supports combined and individual interrupt requests, each of which can be masked. The combined interrupt request is the ORed result of all other DW_apb_ssi interrupts after masking. The system designer has the choice of routing individual interrupt requests or only the combined interrupt request to the Interrupt Controller. All DW_apb_ssi interrupts have the same active polarity level; you can configure this polarity level as active-high or active-low. The DW_apb_ssi interrupts are described as follows:

- ❖ Transmit FIFO Empty Interrupt (ssi_txe_intr) – Set when the transmit FIFO is equal to or below its threshold value and requires service to prevent an under-run. The threshold value, set through a software-programmable register, determines the level of transmit FIFO entries at which an interrupt is generated. This interrupt is cleared by hardware when data are written into the transmit FIFO buffer, bringing it over the threshold level.
- ❖ Transmit FIFO Overflow Interrupt (ssi_txo_intr) – Set when an APB access attempts to write into the transmit FIFO after it has been completely filled. When set, data written from the APB is discarded. This interrupt remains set until you read the transmit FIFO overflow interrupt clear register (TXOICR).

- ❖ Receive FIFO Full Interrupt (`ssi_rxf_intr`) – Set when the receive FIFO is equal to or above its threshold value plus 1 and requires service to prevent an overflow. The threshold value, set through a software-programmable register, determines the level of receive FIFO entries at which an interrupt is generated. This interrupt is cleared by hardware when data are read from the receive FIFO buffer, bringing it below the threshold level.
- ❖ Receive FIFO Overflow Interrupt (`ssi_rxo_intr`) – Set when the receive logic attempts to place data into the receive FIFO after it has been completely filled. When set, newly received data are discarded. This interrupt remains set until you read the receive FIFO overflow interrupt clear register ([RXOICR](#)).
- ❖ Receive FIFO Underflow Interrupt (`ssi_rxu_intr`) – Set when an APB access attempts to read from the receive FIFO when it is empty. When set, zeros are read back from the receive FIFO. This interrupt remains set until you read the receive FIFO underflow interrupt clear register ([RXUICR](#)).
- ❖ Multi-Master Contention Interrupt (`ssi_mst_intr`) – Present only when the DW_apb_ssi component is configured as a serial-master device. The interrupt is set when another serial master on the serial bus selects the DW_apb_ssi master as a serial-slave device and is actively transferring data. This informs the processor of possible contention on the serial bus. This interrupt remains set until you read the multi-master interrupt clear register ([MSTICR](#)).
- ❖ Combined Interrupt Request (`ssi_intr`) – OR'ed result of all the above interrupt requests after masking. To mask this interrupt signal, you must mask all other DW_apb_ssi interrupt requests.

3.2 Transfer Modes

When transferring data on the serial bus, the DW_apb_ssi operates in the modes discussed in this section. The transfer mode (TMOD) is set by writing to control register 0 (CTRLR0), as described in “[CTRLR0](#)” on page 94.



The transfer mode setting does not affect the duplex of the serial transfer. TMOD is ignored for Microwire transfers, which are controlled by the [MWCR](#) register.

3.2.1 Transmit and Receive

When TMOD = 2'b00, both transmit and receive logic are valid. The data transfer occurs as normal according to the selected frame format (serial protocol). Transmit data are popped from the transmit FIFO and sent through the txd line to the target device, which replies with data on the rxd line. The receive data from the target device is moved from the receive shift register into the receive FIFO at the end of each data frame.

3.2.2 Transmit Only

When TMOD = 2'b01, the receive data are invalid and should not be stored in the receive FIFO. The data transfer occurs as normal, according to the selected frame format (serial protocol). Transmit data are popped from the transmit FIFO and sent through the txd line to the target device, which replies with data on the rxd line. At the end of the data frame, the receive shift register does not load its newly received data into the receive FIFO. The data in the receive shift register is overwritten by the next transfer. You should mask interrupts originating from the receive logic when this mode is entered.

3.2.3 Receive Only

When $TMOD = 2'b10$, the transmit data are invalid. When configured as a slave, the transmit FIFO is never popped in Receive Only mode. The txd output remains at a constant logic level during the transmission. The data transfer occurs as normal according to the selected frame format (serial protocol). The receive data from the target device is moved from the receive shift register into the receive FIFO at the end of each data frame. You should mask interrupts originating from the transmit logic when this mode is entered.

3.2.4 EEPROM Read

**Note**

This transfer mode is only valid for master configurations.

When $TMOD = 2'b11$, the transmit data is used to transmit an opcode and/or an address to the EEPROM device. Typically this takes three data frames (8-bit opcode followed by 8-bit upper address and 8-bit lower address). During the transmission of the opcode and address, no data is captured by the receive logic (as long as the DW_apb_ssi master is transmitting data on its txd line, data on the rxd line is ignored). The DW_apb_ssi master continues to transmit data until the transmit FIFO is empty. Therefore, you should ONLY have enough data frames in the transmit FIFO to supply the opcode and address to the EEPROM. If more data frames are in the transmit FIFO than are needed, then read data is lost.

When the transmit FIFO becomes empty (all control information has been sent), data on the receive line (rxd) is valid and is stored in the receive FIFO; the txd output is held at a constant logic level. The serial transfer continues until the number of data frames received by the DW_apb_ssi master matches the value of the NDF field in the [CTRLR1](#) register + 1.

**Note**

EEPROM read mode is not supported when the DW_apb_ssi is configured to be in the SSP mode.

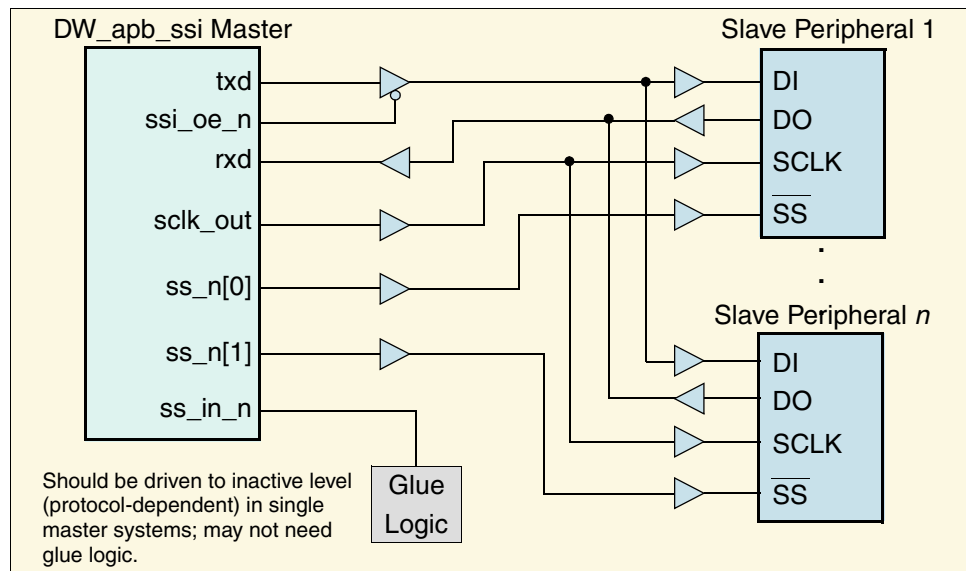
3.3 Operation Modes

The DW_apb_ssi can be configured in the fundamental modes of operation discussed in this section.

3.3.1 Serial-Master Mode

This mode enables serial communication with serial-slave peripheral devices. When configured as a serial-master device, the DW_apb_ssi initiates and controls all serial transfers. Figure 3-4 shows an example of the DW_apb_ssi configured as a serial master with all other devices on the serial bus configured as serial slaves.

Figure 3-4 DW_apb_ssi Configured as Master Device



The serial bit-rate clock, generated and controlled by the DW_apb_ssi, is driven out on the `sclk_out` line. When the DW_apb_ssi is disabled (`SSI_EN = 0`), no serial transfers can occur and `sclk_out` is held in "inactive" state, as defined by the serial protocol under which it operates.

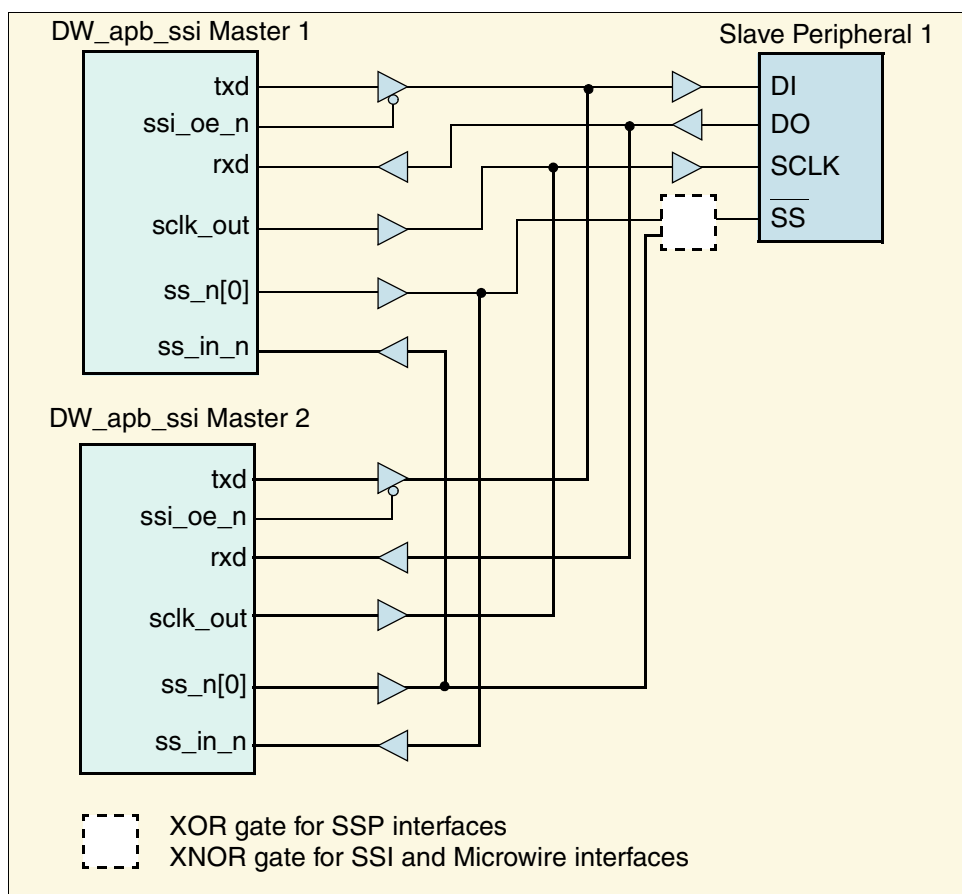
3.3.1.1 Master Contention Input

The DW_apb_ssi master configuration has a serial slave select input, `ss_in_n`, that can be used to inform the DW_apb_ssi master that another serial master is active on the bus. When this input is active – the active level depends on the serial protocol – the DW_apb_ssi master remains in an IDLE state and holds off any pending serial transfer until the `ss_in_n` input is returned to an in-active level.

You should use the `ss_in_n` input to assist arbitration between multiple serial bus masters. A simple usage example is shown in Figure 3-5. In this example, it is a case of "first-come-first-served" arbitration. Although the example does create the potential for locking the bus, if both masters assert their slave select outputs on the same clock edge, it shows how the `ss_in_n` signal can be used. A more complex arbiter block,

that obeys the principles illustrated in the figure, should be used to arbitrate between master select outputs, slave select inputs and master select inputs, to prevent any potential bus locking occurrences.

Figure 3-5 Arbitration Between Multiple Serial Masters



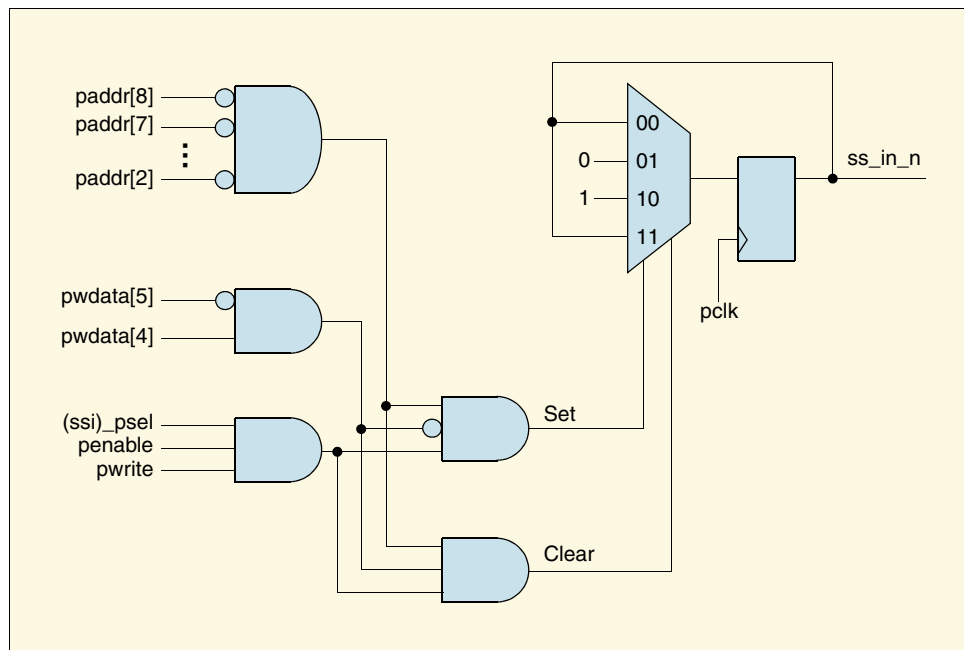
If the DW_apb_ssi master is the only master device on the serial bus, you might need to insert some glue logic to control the logic level on the master ss_in_n input.

Glue logic is required if both of the following are true:

- ❖ You dynamically change the serial protocol
- ❖ One of the protocols being used is SSP

There are several methods for implementing this glue logic; [Figure 3-6](#) illustrates an example architecture.

Figure 3-6 Glue Logic for Controlling Logic Level on Master ss_in_n Input



In this architecture, the `ss_in_n` signal is driven low when the user writes 2'b01 (SSP) into the FRF bit field in control register 0 (CTRLR0). The `ss_in_n` signal is driven high for all other values written into the FRF bit field.



Note If the Default Frame Format in the DW_apb_ssi is not SSP, the register shown in the diagram below should reset to 1.

Glue logic is not required under either of the following conditions:

- ❖ If you never intend to dynamically change the serial protocol that the DW_apb_ssi master is operating under.
- ❖ If you do change the serial protocol dynamically but do not use the SSP protocol.

Under these conditions, the `ss_in_n` signal can be tied high or low depending on which serial protocol you use.

- ❖ If the serial protocol is SPI or MicroWire, the `ss_in_n` signal should be tied high.
- ❖ If the serial protocol being used is SSP, the `ss_in_n` signal should be tied low.

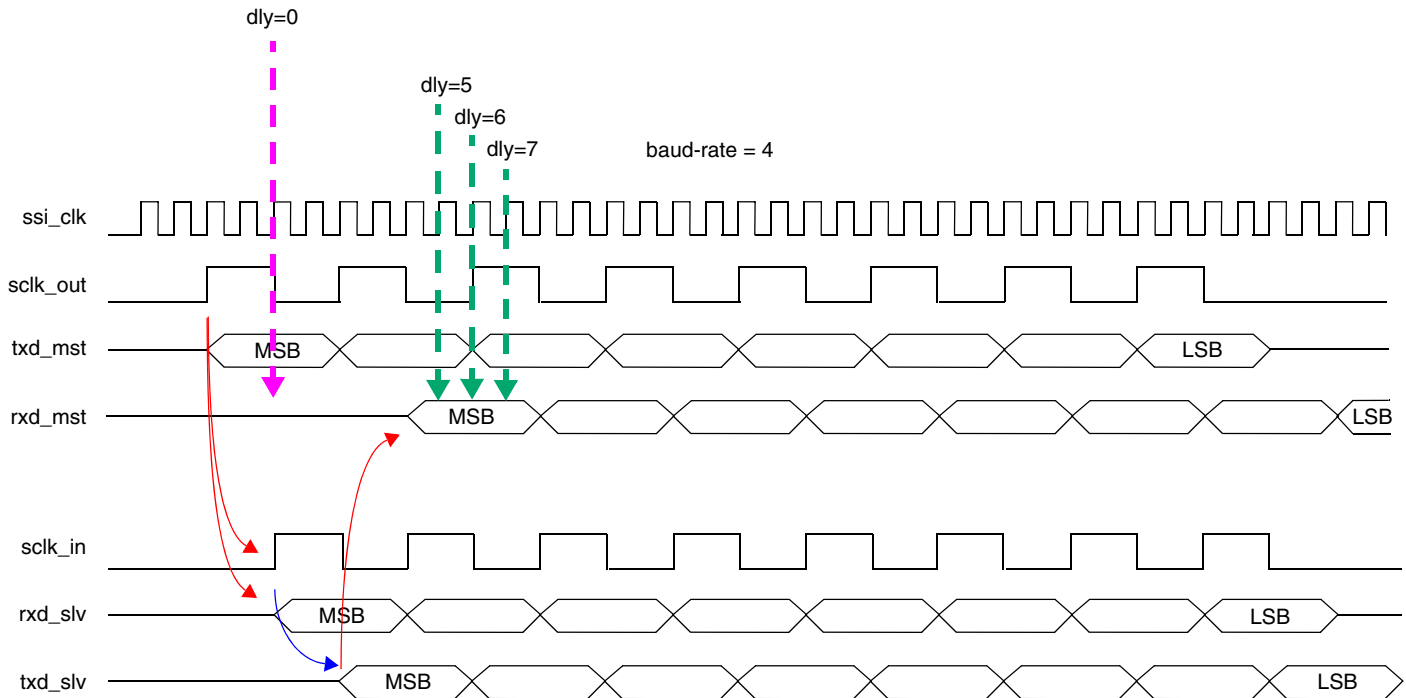
3.3.1.2 RXD Sample Delay

When the DW_apb_ssi is configured as a master, additional logic can be included in the design in order to delay the default sample time of the `rx_d` signal. This additional logic can help to increase the maximum achievable frequency on the serial bus.

To include this additional logic, the `SSI_HAS_RX_SAMPLE_DELAY` parameter should have a value of 1.

Round trip routing delays on the `sclk_out` signal from the master and the `rxd` signal from the slave can mean that the timing of the `rxd` signal – as seen by the master – has moved away from the normal sampling time. Figure 3-7 illustrates this situation.

Figure 3-7 Effects of Round-Trip Routing Delays on `sclk_out` Signal



Red arrows indicate routing delay between master and slave devices

Blue arrow indicates sampling delay within slave from receiving `slk_in` to driving `txd_out`

The Slave uses the `sclk_out` signal from the master as a strobe in order to drive `rxd` signal data onto the serial bus. Routing and sampling delays on the `sclk_out` signal by the slave device can mean that the `rxd` bit has not stabilized to the correct value before the master samples the `rxd` signal. Figure 3-7 shows an example of how a routing delay on the `rxd` signal can result in an incorrect `rxd` value at the default time when the master samples the port.

Without the RXD Sample Delay logic, the user would have to increase the baud-rate for the transfer in order to ensure that the setup times on the `rxd` signal are within range; this results in reducing the frequency of the serial interface.

When the RXD Sample Delay logic is included, the user can dynamically program a delay value in order to move the sampling time of the `rxd` signal equal to a number of `ssi_clk` cycles from the default.

The sample delay logic has a resolution of one `ssi_clk` cycle. Software can “train” the serial bus by coding a loop that continually reads from the slave and increments the master's RXD Sample Delay value until the correct data is received by the master.

3.3.1.3 Data Transfers

Data transfers are started by the serial-master device. When the `DW_apb_ssi` is enabled (`SSI_EN=1`), at least one valid data entry is present in the transmit FIFO and a serial-slave device is selected. When actively transferring data, the busy flag (`BUSY`) in the status register (`SR`) is set. You must wait until the busy flag is cleared before attempting a new serial transfer.

**Note**

The BUSY status is not set when the data are written into the transmit FIFO. This bit gets set only when the target slave has been selected and the transfer is underway. After writing data into the transmit FIFO, the shift logic does not begin the serial transfer until a positive edge of the `sclk_out` signal is present. The delay in waiting for this positive edge depends on the baud rate of the serial transfer. Before polling the BUSY status, you should first poll the TFE status (waiting for 1) or wait for $\text{BAUDR} * \text{ssi_clk}$ clock cycles.

3.3.1.4 Master SPI and SSP Serial Transfers

The sections “[Motorola Serial Peripheral Interface \(SPI\)](#)” on page 49 and “[Texas Instruments Synchronous Serial Protocol \(SSP\)](#)” on page 54 describe the SPI and SSP serial protocols, respectively. They include timing diagrams and provide information as to how data are structured in the transmit and receive the FIFOs before and after the serial transfer.

When the transfer mode is “transmit and receive” or “transmit only” ($\text{TMOD} = 2'b00$ or $\text{TMOD} = 2'b01$, respectively), transfers are terminated by the shift control logic when the transmit FIFO is empty. For continuous data transfers, you must ensure that the transmit FIFO buffer does not become empty before all the data have been transmitted. The transmit FIFO threshold level (TXFTLR) can be used to early interrupt (`ssi_txe_intr`) the processor indicating that the transmit FIFO buffer is nearly empty.

When a DMA is used for APB accesses, the transmit data level (DMATDLR) can be used to early request (`dma_tx_req`) the DMA Controller, indicating that the transmit FIFO is nearly empty. The FIFO can then be refilled with data to continue the serial transfer. The user may also write a block of data (at least two FIFO entries) into the transmit FIFO before enabling a serial slave. This ensures that serial transmission does not begin until the number of data-frames that make up the continuous transfer are present in the transmit FIFO.

When the transfer mode is “receive only” ($\text{TMOD} = 2'b10$), a serial transfer is started by writing one “dummy” data word into the transmit FIFO when a serial slave is selected. The `txd` output from the DW_apb_ssi is held at a constant logic level for the duration of the serial transfer. The transmit FIFO is popped only once at the beginning and may remain empty for the duration of the serial transfer. The end of the serial transfer is controlled by the “number of data frames” (NDF) field in control register 1 ([CTRLR1](#)).

If, for example, you want to receive 24 data frames from a serial-slave peripheral, you should program the NDF field with the value 23; the receive logic terminates the serial transfer when the number of frames received is equal to the NDF value + 1. This transfer mode increases the bandwidth of the APB bus as the transmit FIFO never needs to be serviced during the transfer. The receive FIFO buffer should be read each time the receive FIFO generates a FIFO full interrupt request to prevent an overflow.

When the transfer mode is “eeprom_read” ($\text{TMOD} = 2'b11$), a serial transfer is started by writing the opcode and/or address into the transmit FIFO when a serial slave (EEPROM) is selected. The opcode and address are transmitted to the EEPROM device, after which read data is received from the EEPROM device and stored in the receive FIFO. The end of the serial transfer is controlled by the NDF field in the control register 1 ([CTRLR1](#)).

**Note**

EEPROM read mode is not supported when the DW_apb_ssi is configured to be in the SSP mode.

The receive FIFO threshold level (RXFTLR) can be used to give early indication that the receive FIFO is nearly full. When a DMA is used for APB accesses, the receive data level (DMARDLR) can be used to early request (`dma_rx_req`) the DMA Controller, indicating that the receive FIFO is nearly full.

A typical software flow for completing an SPI or SSP serial transfer from the DW_apb_ssi serial master is outlined as follows:

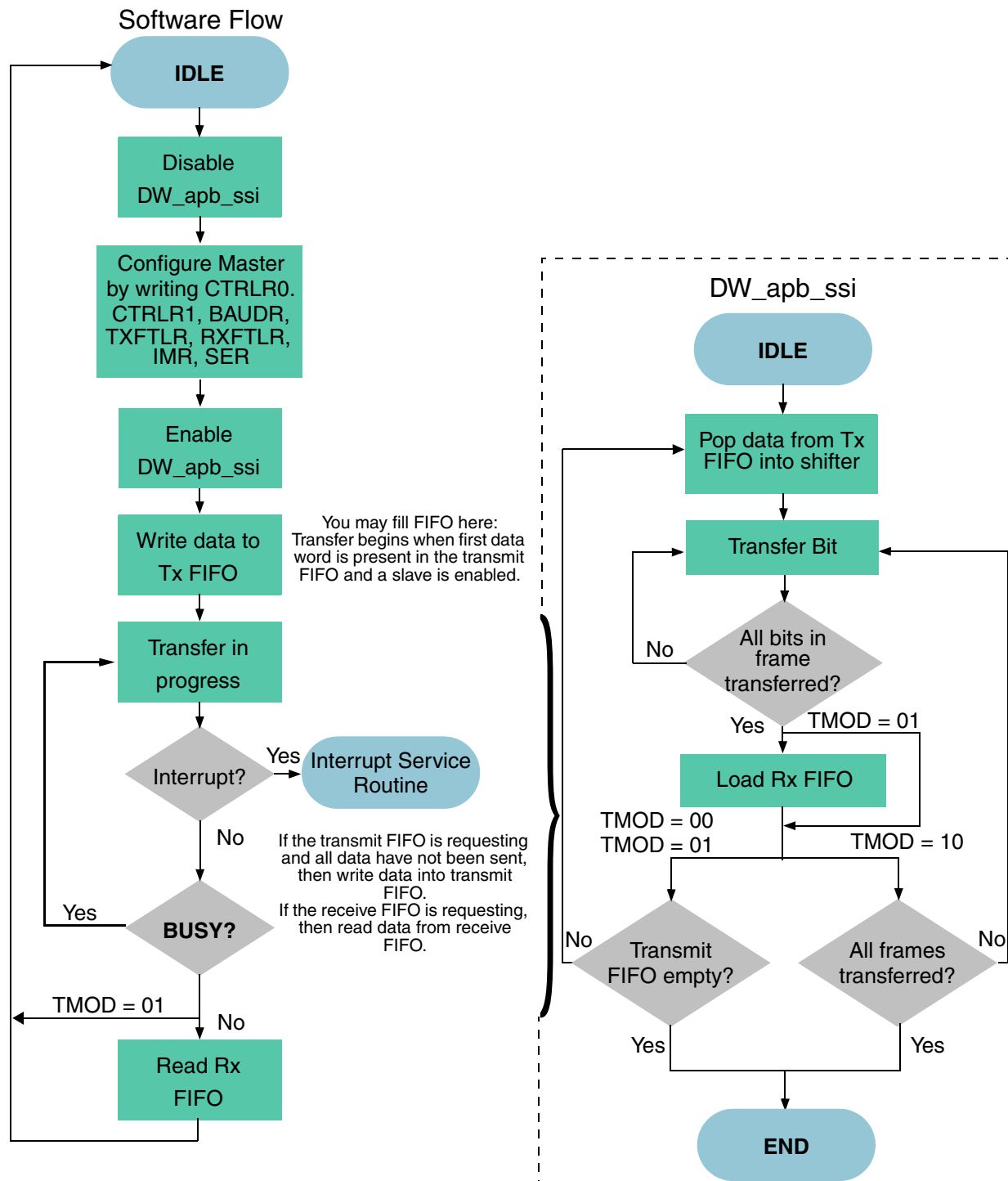
1. If the DW_apb_ssi is enabled, disable it by writing 0 to the SSI Enable register ([SSIENR](#)).
2. Set up the DW_apb_ssi control registers for the transfer; these registers can be set in any order.
 - ◆ Write Control Register 0 ([CTRLR0](#)). For SPI transfers, the serial clock polarity and serial clock phase parameters must be set identical to target slave device.
 - ◆ If the transfer mode is *receive only*, write [CTRLR1](#) (Control Register 1) with the number of frames in the transfer minus 1; for example, if you want to receive four data frames, write this register with 3.
 - ◆ Write the Baud Rate Select Register ([BAUDR](#)) to set the baud rate for the transfer.
 - ◆ Write the Transmit and Receive FIFO Threshold Level registers ([TXFTLR](#) and [RXFTLR](#), respectively) to set FIFO threshold levels.
 - ◆ Write the [IMR](#) register to set up interrupt masks.
 - ◆ The Slave Enable Register ([SER](#)) register can be written here to enable the target slave for selection. If a slave is enabled here, the transfer begins as soon as one valid data entry is present in the transmit FIFO. If no slaves are enabled prior to writing to the Data Register ([DR](#)), the transfer does not begin until a slave is enabled.
3. Enable the DW_apb_ssi by writing 1 to the [SSIENR](#) register.
4. Write data for transmission to the target slave into the transmit FIFO (write [DR](#)).

If no slaves were enabled in the [SER](#) register at this point, enable it now to begin the transfer.
5. Poll the BUSY status to wait for completion of the transfer. The BUSY status cannot be polled immediately; for more information, see the note on [page 40](#).

If a transmit FIFO empty interrupt request is made, write the transmit FIFO (write [DR](#)). If a receive FIFO full interrupt request is made, read the receive FIFO (read [DR](#)).
6. The transfer is stopped by the shift control logic when the transmit FIFO is empty. If the transfer mode is *receive only* (TMOD = 2'b10), the transfer is stopped by the shift control logic when the specified number of frames have been received. When the transfer is done, the BUSY status is reset to 0.
7. If the transfer mode is not *transmit only* (TMOD != 01), read the receive FIFO until it is empty.
8. Disable the DW_apb_ssi by writing 0 to [SSIENR](#).

Figure 3-8 shows a typical software flow for starting a DW_apb_ssi master SPI/SSP serial transfer. The diagram also shows the hardware flow inside the serial-master component.

Figure 3-8 DW_apb_ssi Master SPI/SSP Transfer Flow



3.3.1.5 Master Microwire Serial Transfers

“National Semiconductor Microwire” on page 55 describes the Microwire serial protocol in detail, including timing diagrams and explaining how data are structured in the transmit and receive FIFOs before and after a serial transfer.

Microwire serial transfers from the DW_apb_ssi serial master are controlled by the Microwire Control Register (**MWCR**). The MWHS bit field enables and disables the Microwire handshaking interface. The MDD bit field controls the direction of the data frame (the control frame is always transmitted by the master and received by the slave). The MWMOD bit field defines whether the transfer is sequential or nonsequential.

All Microwire transfers are started by the DW_apb_ssi serial master when there is at least one control word in the transmit FIFO and a slave is enabled. When the DW_apb_ssi master transmits the data frame (MDD = 1), the transfer is terminated by the shift logic when the transmit FIFO is empty. When the DW_apb_ssi master receives the data frame (MDD = 1), the termination of the transfer depends on the setting of the MWMOD bit field. If the transfer is nonsequential (MWMOD = 0), it is terminated when the transmit FIFO is empty after shifting in the data frame from the slave. When the transfer is sequential (MWMOD = 1), it is terminated by the shift logic when the number of data frames received is equal to the value in the **CTRLR1** register + 1.

When the handshaking interface on the DW_apb_ssi master is enabled (MWHS = 1), the status of the target slave is polled after transmission. Only when the slave reports a *ready status* does the DW_apb_ssi master complete the transfer and clear its BUSY status. If the transfer is continuous, the next control/data frame is not sent until the slave device returns a *ready status*.

A typical software flow for completing a Microwire serial transfer from the DW_apb_ssi serial master is outlined as follows:

1. If the DW_apb_ssi is enabled, disable it by writing 0 to **SSIENR**.
2. Set up the DW_apb_ssi control registers for the transfer. These registers can be set in any order. Write **CTRLR0** to set transfer parameters.
 - ◆ If the transfer is sequential and the DW_apb_ssi master receives data, write **CTRLR1** with the number of frames in the transfer minus 1; for instance, if you want to receive four data frames, write this register with 3.
 - ◆ Write **BAUDR** to set the baud rate for the transfer.
 - ◆ Write **TXFTLR** and **RXFTLR** to set FIFO threshold levels.
 - ◆ Write the **IMR** register to set up interrupt masks.

You can write the **SER** register to enable the target slave for selection. If a slave is enabled here, the transfer begins as soon as one valid data entry is present in the transmit FIFO. If no slaves are enabled prior to writing to the **DR** register, the transfer does not begin until a slave is enabled.

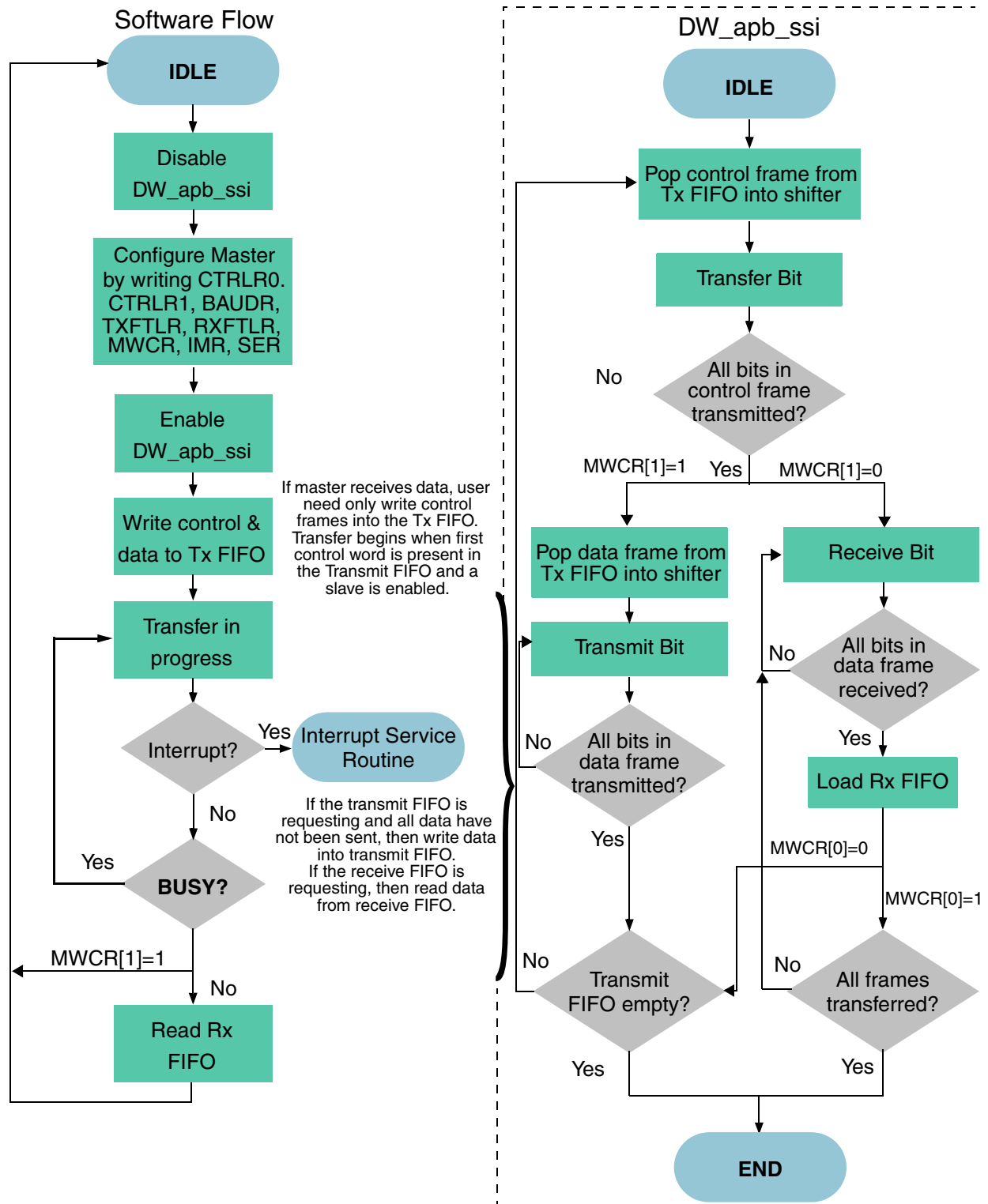
3. Enable the DW_apb_ssi by writing 1 to the **SSIENR** register.
4. If the DW_apb_ssi master transmits data, write the control and data words into the transmit FIFO (write **DR**). If the DW_apb_ssi master receives data, write the control word(s) into the transmit FIFO. If no slaves were enabled in the **SER** register at this point, enable now to begin the transfer.
5. Poll the BUSY status to wait for completion of the transfer. The BUSY status cannot be polled immediately; for more information, see the note on page 40.

If a transmit FIFO empty interrupt request is made, write the transmit FIFO (write **DR**). If a receive FIFO full interrupt request is made, read the receive FIFO (read **DR**).

6. The transfer is stopped by the shift control logic when the transmit FIFO is empty. If the transfer mode is sequential and the DW_apb_ssi master receives data, the transfer is stopped by the shift control logic when the specified number of data frames is received. When the transfer is done, the BUSY status is reset to 0.
7. If the DW_apb_ssi master receives data, read the receive FIFO until it is empty.
8. Disable the DW_apb_ssi by writing 0 to [SSIENR](#).

Figure 3-9 shows a typical software flow for starting a DW_apb_ssi master Microwire serial transfer. The diagram also shows the hardware flow inside the serial-master component.

Figure 3-9 DW_apb_ssi Master Microwire Transfer Flow

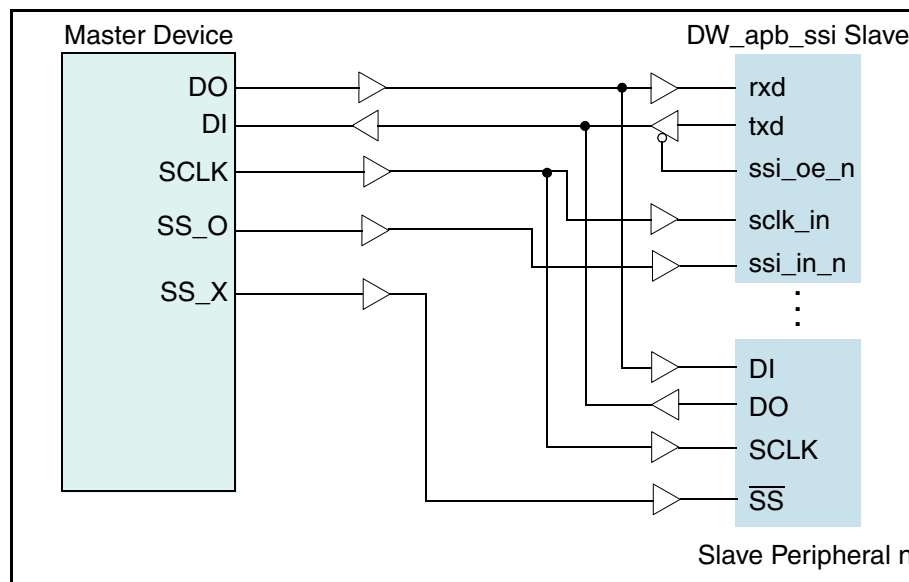


3.3.2 Serial-Slave Mode

This mode enables serial communication with master peripheral devices. When the DW_apb_ssi is configured as a slave device, all serial transfers are initiated and controlled by the serial bus master. [Figure 3-10](#) shows an example of the DW_apb_ssi configured as a serial slave in a single-master bus system.

When the DW_apb_ssi serial slave is selected during configuration, it enables its txd data onto the serial bus. All data transfers to and from the serial slave are regulated on the serial clock line (sclk_in), driven from the serial-master device. Data are propagated from the serial slave on one edge of the serial clock line and sampled on the opposite edge.

Figure 3-10 DW_apb_ssi Configured as Slave Device



When the DW_apb_ssi serial slave is not selected, it must not interfere with data transfers between the serial-master and other serial-slave devices. When the serial slave is not selected, its txd output is buffered, resulting in a high impedance drive onto the serial master rxd line. The buffers shown in [Figure 3-10](#) are external to DW_apb_ssi.

The serial clock that regulates the data transfer is generated by the serial-master device and input to the DW_apb_ssi slave on sclk_in. The slave remains in an idle state until selected by the bus master. When not actively transmitting data, the slave must hold its txd line in a high impedance state to avoid interference with serial transfers to other slave devices. The ssi_oe_n line is available for use to control the txd output buffer. The slave continues to transfer data to and from the master device as long as it is selected. If the master transmits to all serial slaves, a control bit (SLV_OE) in the DW_apb_ssi control register 0 ([CTRLR0](#)) can be programmed to inform the slave if it should respond with data from the its txd line.

3.3.2.1 Slave SPI and SSP Serial Transfers

“[Motorola Serial Peripheral Interface \(SPI\)](#)” on page 49 and “[Texas Instruments Synchronous Serial Protocol \(SSP\)](#)” on page 54 contain a description of the SPI and SSP serial protocols, respectively. The sections also provide timing diagrams and information on how data are structured in the transmit and receive FIFOs before and after the serial transfer.

If the DW_apb_ssi slave is *receive only* (TMOD=10), the transmit FIFO need not contain valid data because the data currently in the transmit shift register is resent each time the slave device is selected. The TXE error

flag in the status register ([SR](#)) is not set when TMOD=01. You should mask the transmit FIFO empty interrupt when this mode is used.

If the DW_apb_ssi slave transmits data to the master, you must ensure that data exists in the transmit FIFO before a transfer is initiated by the serial-master device. If the master initiates a transfer to the DW_apb_ssi slave when no data exists in the transmit FIFO, an error flag (TXE) is set in the DW_apb_ssi status register, and the previously transmitted data frame is resent on txd. For continuous data transfers, you must ensure that the transmit FIFO buffer does not become empty before all the data have been transmitted. The transmit FIFO threshold level register ([TXFTLR](#)) can be used to early interrupt (ssi_txe_intr) the processor, indicating that the transmit FIFO buffer is nearly empty. When a DMA Controller is used for APB accesses, the DMA transmit data level register ([DMATDLR](#)) can be used to early request (dma_tx_req) the DMA Controller, indicating that the transmit FIFO is nearly empty. The FIFO can then be refilled with data to continue the serial transfer.

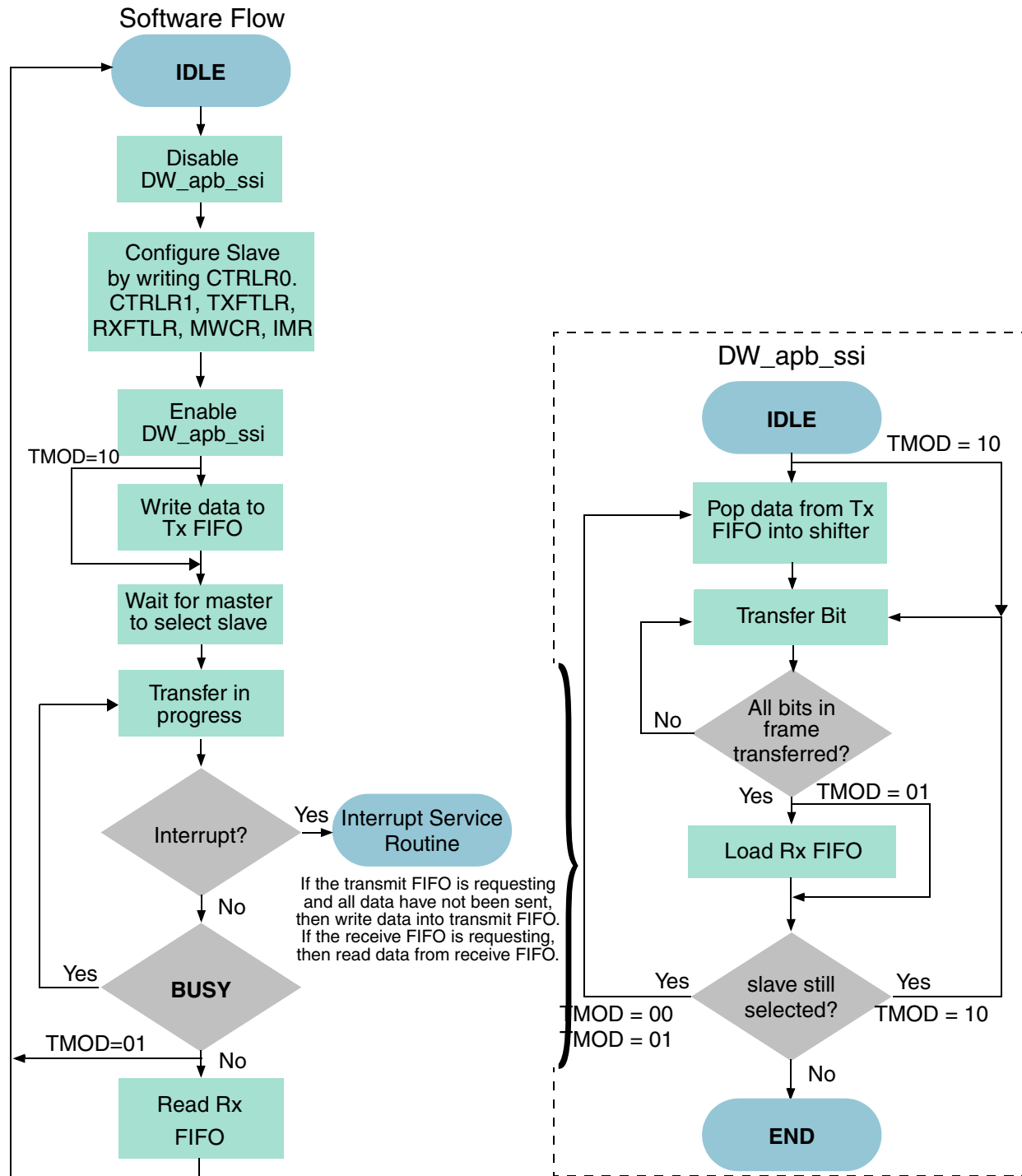
The receive FIFO buffer should be read each time the receive FIFO generates a FIFO full interrupt request to prevent an overflow. The receive FIFO threshold level register ([RXFTLR](#)) can be used to give early indication that the receive FIFO is nearly full. When a DMA Controller is used for APB accesses, the DMA receive data level register ([DMARDLR](#)) can be used to early request (dma_rx_req) the DMA controller, indicating that the receive FIFO is nearly full.

A typical software flow for completing a continuous serial transfer from a serial master to the DW_apb_ssi slave is described as follows:

1. If the DW_apb_ssi is enabled, disable it by writing 0 to [SSIENR](#).
2. Set up the DW_apb_ssi control registers for the transfer. These registers can be set in any order.
 - a. Write [CTRLR0](#) (for SPI transfers SCPH and SCPOL must be set identical to the master device).
 - b. Write [TXFTLR](#) and [RXFTLR](#) to set FIFO threshold levels.
 - c. Write the [IMR](#) register to set up interrupt masks.
3. Enable the DW_apb_ssi by writing 1 to the [SSIENR](#) register.
4. If the transfer mode is *transmit and receive* (TMOD=2'b00) or *transmit only* (TMOD=2'b01), write data for transmission to the master into the transmit FIFO (Write [DR](#)).
 If the transfer mode is *receive only* (TMOD=2'b10), there is no need to write data into the transmit FIFO; the current value in the transmit shift register is retransmitted.
5. The DW_apb_ssi slave is now ready for the serial transfer. The transfer begins when the DW_apb_ssi slave is selected by a serial-master device.
6. When the transfer is underway, the BUSY status can be polled to return the transfer status. If a transmit FIFO empty interrupt request is made, write the transmit FIFO (write [DR](#)). If a receive FIFO full interrupt request is made, read the receive FIFO (read [DR](#)).
7. The transfer ends when the serial master removes the select input to the DW_apb_ssi slave. When the transfer is completed, the BUSY status is reset to 0.
8. If the transfer mode is not *transmit only* (TMOD != 01), read the receive FIFO until empty.
9. Disable the DW_apb_ssi by writing 0 to [SSIENR](#).

Figure 3-11 shows a typical software flow for a DW_apb_ssi slave SPI or SSP serial transfer. The diagram also shows the hardware flow inside the serial-slave component.

Figure 3-11 DW_apb_ssi Slave SPI/SSP Transfer Flow



3.3.2.2 Slave Microwire Serial Transfers

“National Semiconductor Microwire” on page 55 describes the Microwire serial protocol in detail, including timing diagrams and information on how data are structured in the transmit and receive FIFOs before and after a serial transfer. When the DW_apb_ssi is configured as a slave device, the Microwire protocol operates in much the same way as the SPI protocol. There is no decode of the control frame by the DW_apb_ssi slave device.

3.4 Partner Connection Interfaces

The DW_apb_ssi can connect to any serial-master or serial-slave peripheral device using one of the interfaces discussed in the following sections.

3.4.1 Motorola Serial Peripheral Interface (SPI)

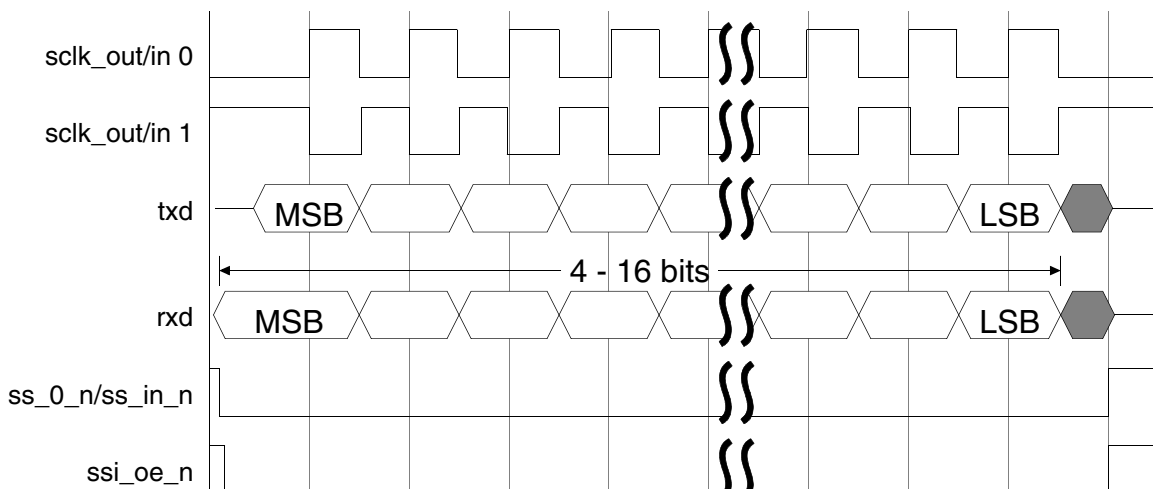
With the SPI, the clock polarity (SCPOL) configuration parameter determines whether the inactive state of the serial clock is high or low. To transmit data, both SPI peripherals must have identical serial clock phase (SCPH) and clock polarity (SCPOL) values. The data frame can be 4 to 16 bits in length.

When the configuration parameter SCPH = 0, data transmission begins on the falling edge of the slave select signal. The first data bit is captured by the master and slave peripherals on the first edge of the serial clock; therefore, valid data must be present on the txd and rxd lines prior to the first serial clock edge. Figure 3-12 shows a timing diagram for a single SPI data transfer with SCPH = 0. The serial clock is shown for configuration parameters SCPOL = 0 and SCPOL = 1.

The following signals are illustrated in the timing diagrams in this section:

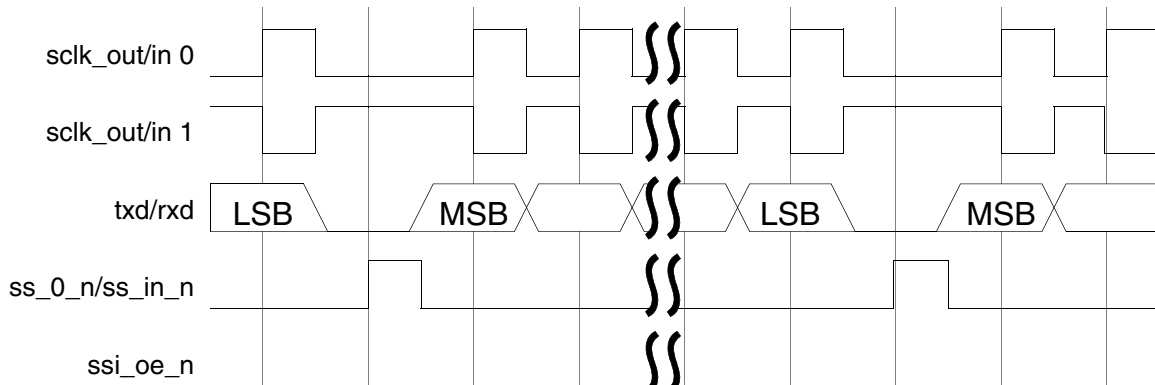
- ❖ sclk_out – serial clock from DW_apb_ssi master (master configuration only)
- ❖ sclk_in – serial clock from DW_apb_ssi slave (slave configuration only)
- ❖ ss_0_n – slave select signal from DW_apb_ssi master (master configuration only)
- ❖ ss_in_n – slave select input to the DW_apb_ssi slave
- ❖ ss_oe_n – output enable for the DW_apb_ssi master/slave
- ❖ txd – transmit data line for the DW_apb_ssi master/slave
- ❖ rxd – receive data line for the DW_apb_ssi master/slave

Figure 3-12 SPI Serial Format (SCPH = 0)



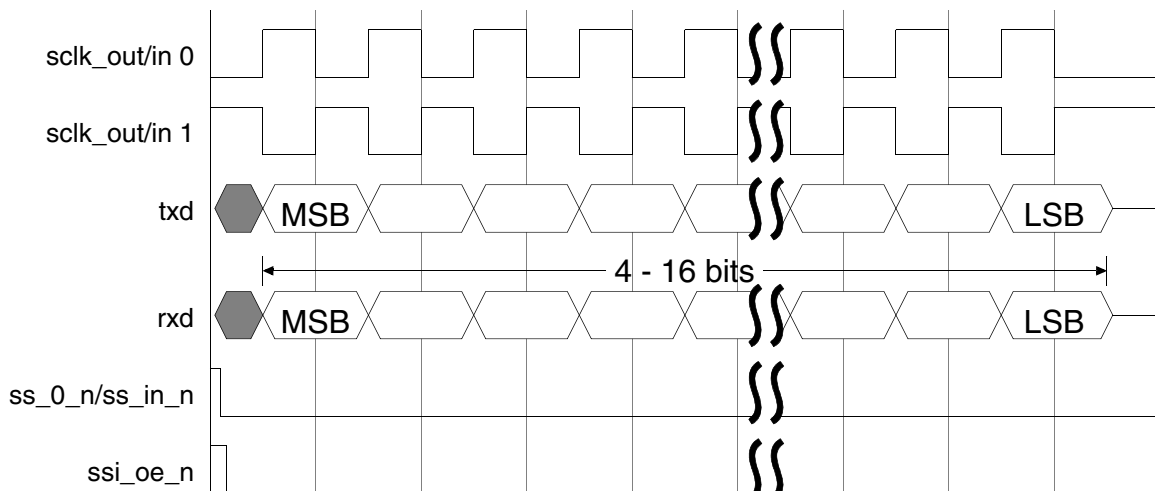
As data transmission starts on the falling edge of the slave select signal when $SCPH = 0$, continuous data transfers require the slave select signal to toggle before beginning the next data frame. This is illustrated in [Figure 3-13](#).

Figure 3-13 SPI Serial Format Continuous Transfers ($SCPH = 0$)



When the configuration parameter $SCPH = 1$, both master and slave peripherals begin transmitting data on the first serial clock edge after the slave select line is activated. The first data bit is captured on the second (trailing) serial clock edge. Data are propagated by the master and slave peripherals on the leading edge of the serial clock. During continuous data frame transfers, the slave select line may be held active-low until the last bit of the last frame has been captured. [Figure 3-14](#) shows the timing diagram for the SPI format when the configuration parameter $SCPH = 1$.

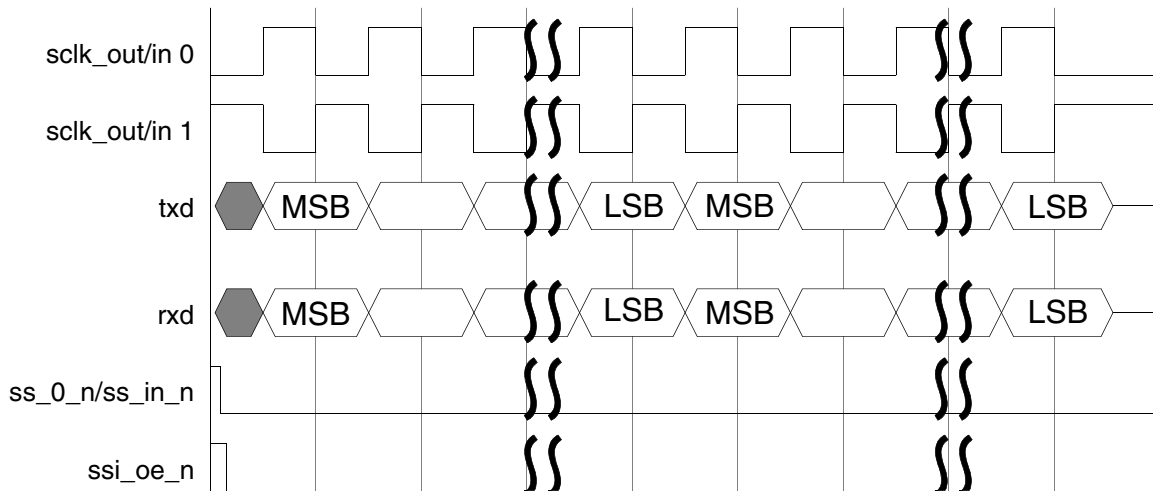
Figure 3-14 SPI Serial Format ($SCPH = 1$)



Continuous data frames are transferred in the same way as single frames, with the MSB of the next frame following directly after the LSB of the current frame. The slave select signal is held active for the duration of

the transfer. [Figure 3-15](#) shows the timing diagram for continuous SPI transfers when the configuration parameter $SCPH = 1$.

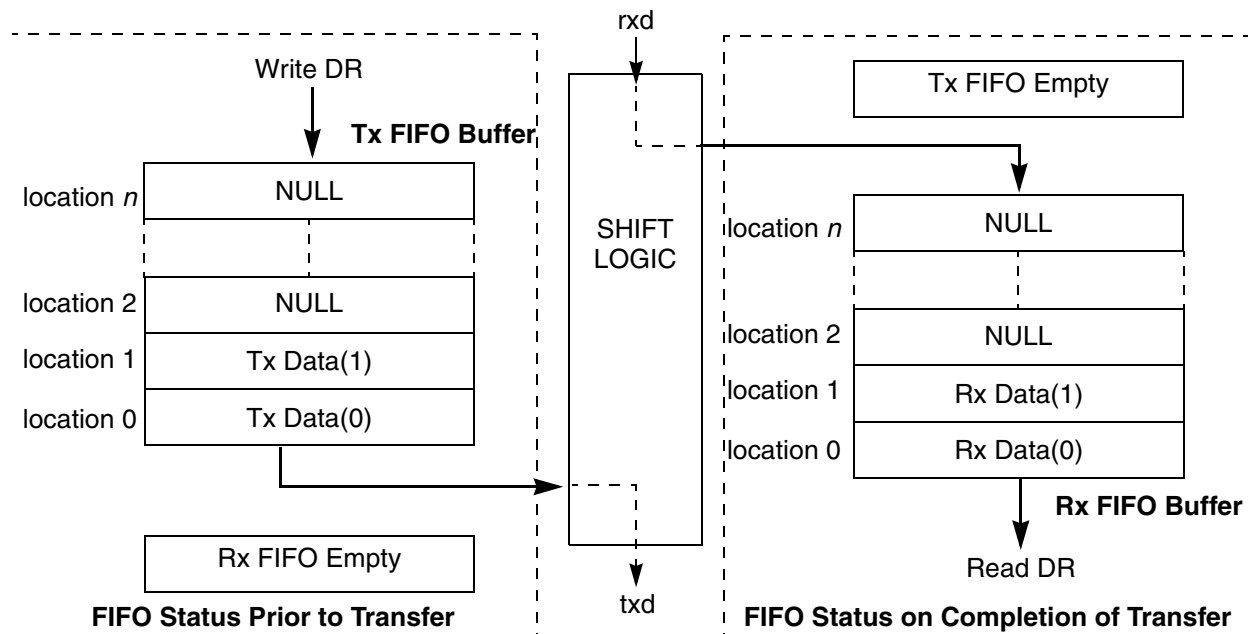
Figure 3-15 SPI Serial Format Continuous Transfer ($SCPH = 1$)



There are four possible transfer modes on the DW_apb_ssi for performing SPI serial transactions; see [“Transfer Modes”](#) on page 34. For *transmit and receive transfers* (transfer mode field (9:8) of the Control Register 0 = 2'b00), data transmitted from the DW_apb_ssi to the external serial device is written into the transmit FIFO. Data received from the external serial device into the DW_apb_ssi is pushed into the receive FIFO.

[Figure 3-16](#) shows the FIFO levels prior to the beginning of a serial transfer and the FIFO levels on completion of the transfer. In this example, two data words are transmitted from the DW_apb_ssi to the external serial device in a continuous transfer. The external serial device also responds with two data words for the DW_apb_ssi.

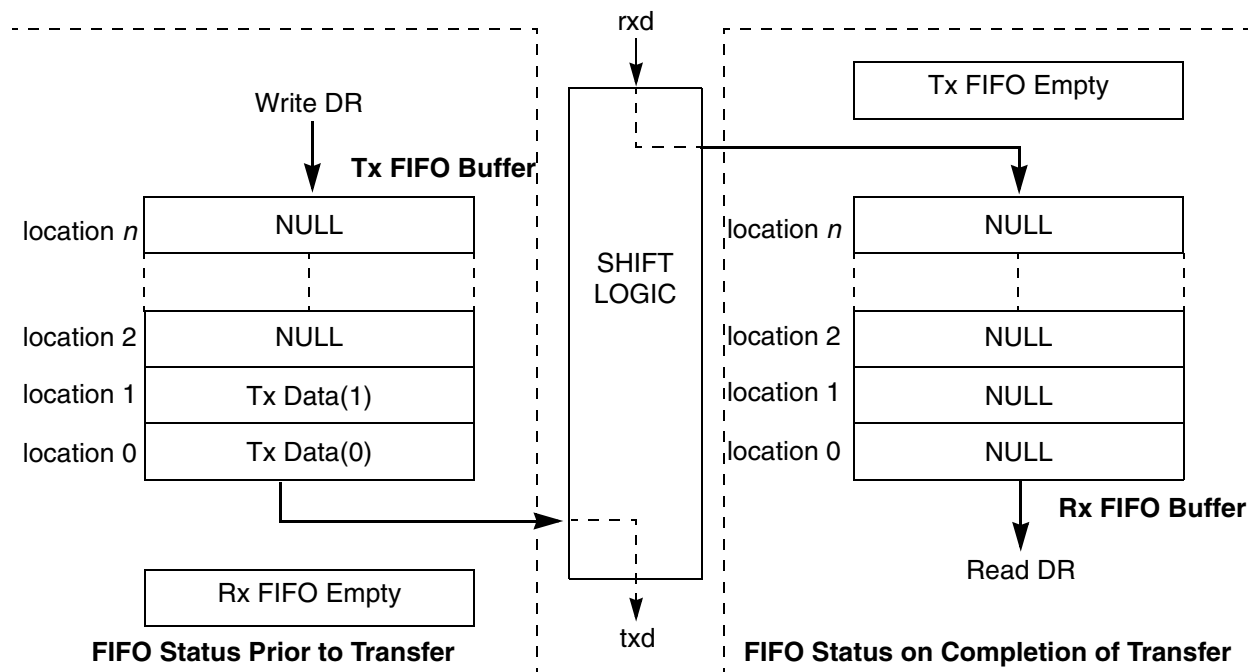
Figure 3-16 FIFO Status for *Transmit & Receive* SPI and SSP Transfers



For *transmit only* transfers (transfer mode field (9:8) of the Control Register 0 = 2'b01), data transmitted from the DW_apb_ssi to the external serial device is written into the transmit FIFO. As the data received from the external serial device is deemed invalid, it is not stored in the DW_apb_ssi receive FIFO.

Figure 3-17 shows the FIFO levels prior to the beginning of a serial transfer and the FIFO levels on completion of the transfer. In this example, two data words are transmitted from the DW_apb_ssi to the external serial device in a continuous transfer.

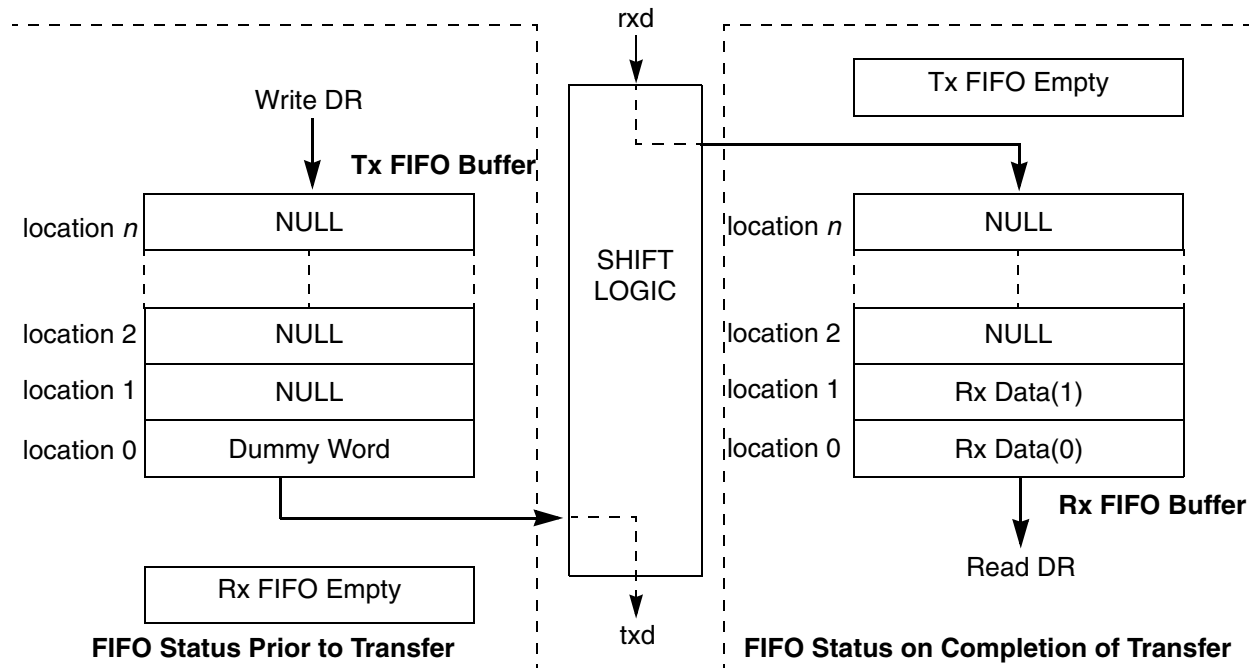
Figure 3-17 FIFO Status for *Transmit Only* SPI and SSP Transfers



For *receive only* transfers (transfer mode field (9:8) of the Control Register 0 = 2'b10), data transmitted from the DW_apb_ssi to the external serial device is invalid, so a single dummy word is written into the transmit FIFO to begin the serial transfer. The txd output from the DW_apb_ssi is held at a constant logic level for the duration of the serial transfer. Data received from the external serial device into the DW_apb_ssi is pushed into the receive FIFO.

Figure 3-18 shows the FIFO levels prior to the beginning of a serial transfer and the FIFO levels on completion of the transfer. In this example, two data words are received by the DW_apb_ssi from the external serial device in a continuous transfer.

Figure 3-18 FIFO Status for *Receive Only* SPI and SSP Transfers

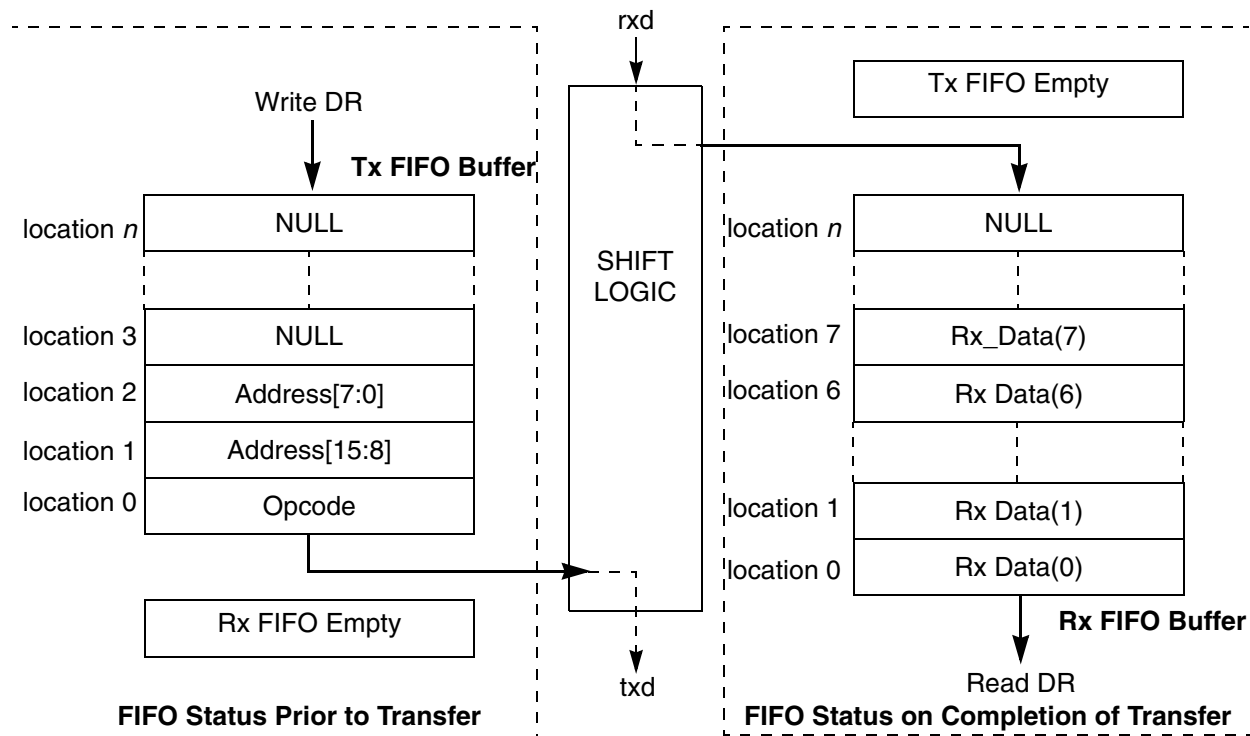


For eeprom_read transfers (transfer mode field [9:8] of the Control Register 0 = 2'b11), opcode and/or EEPROM address are written into the transmit FIFO. During transmission of these control frames, received data is not captured by the DW_apb_ssi master. After the control frames have been transmitted, receive data from the EEPROM is stored in the receive FIFO.

Figure 3-19 shows the FIFO levels prior to the beginning of a serial transfer and the FIFO levels on completion of the transfer. In this example, one opcode and an upper and lower address are transmitted to

the EEPROM, and eight data frames are read from the EEPROM and stored in the receive FIFO of the DW_apb_ssi master.

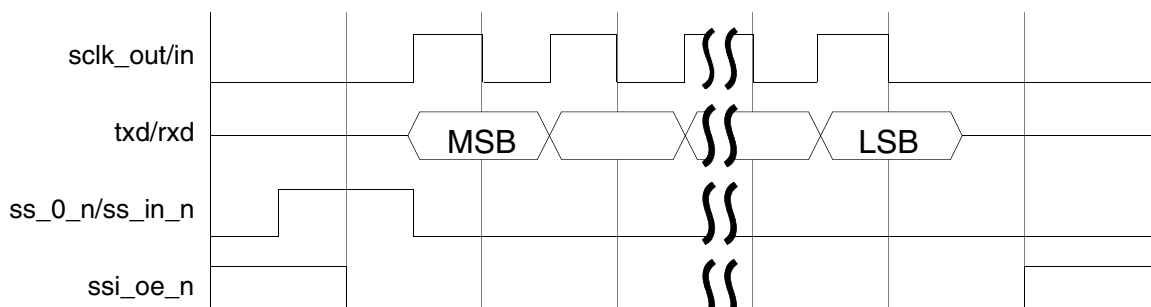
Figure 3-19 FIFO Status for *EEPROM* Read Transfer Mode



3.4.2 Texas Instruments Synchronous Serial Protocol (SSP)

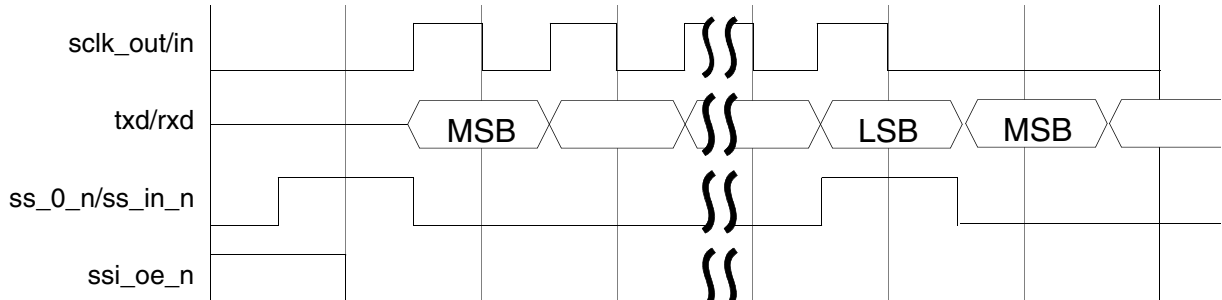
Data transfers begin by asserting the frame indicator line (ss_0_n/ss_in_n) for one serial clock period. Data to be transmitted are driven onto the txd line one serial clock cycle later; similarly data from the slave are driven onto the rxd line. Data are propagated on the rising edge of the serial clock ($sclk_out/sclk_in$) and captured on the falling edge. The length of the data frame ranges from 4 to 16 bits. [Figure 3-20](#) shows the timing diagram for a single SSP serial transfer.

Figure 3-20 SSP Serial Format



Continuous data frames are transferred in the same way as single data frames. The frame indicator is asserted for one clock period during the same cycle as the LSB from the current transfer, indicating that another data frame follows. [Figure 3-21](#) shows the timing for a continuous SSP transfer.

Figure 3-21 SSP Serial Format Continuous Transfer



3.4.3 National Semiconductor Microwire

When the DW_apb_ssi is configured as a serial master, data transmission begins with the falling edge of the slave-select signal (*ss_0_n*). One-half serial clock (*sclk_out*) period later, the first bit of the control is sent out on the *txd* line. The length of the control word can be in the range 1 to 16 bits and is set by writing bit field CFS (bits 15:12) in [CTRLR0](#). The remainder of the control word is transmitted (propagated on the falling edge of *sclk_out*) by the DW_apb_ssi serial master. During this transmission, no data are present (high impedance) on the serial master's *rx*d line.

The direction of the data word is controlled by the MDD bit field (bit 1) in the Microwire Control Register ([MWCR](#)). When MDD=0, this indicates that the DW_apb_ssi serial master receives data from the external serial slave. One clock cycle after the LSB of the control word is transmitted, the slave peripheral responds with a dummy 0 bit, followed by the data frame, which can be 4 to 16 bits in length. Data are propagated on the falling edge of the serial clock and captured on the rising edge.

The slave-select signal is held active-low during the transfer and is de-asserted one-half clock cycle later, after the data are transferred. [Figure 3-22](#) shows the timing diagram for a single DW_apb_ssi serial master read from an external serial slave.

Figure 3-22 Single DW_apb_ssi Master Microwire Serial Transfer (MDD=0)

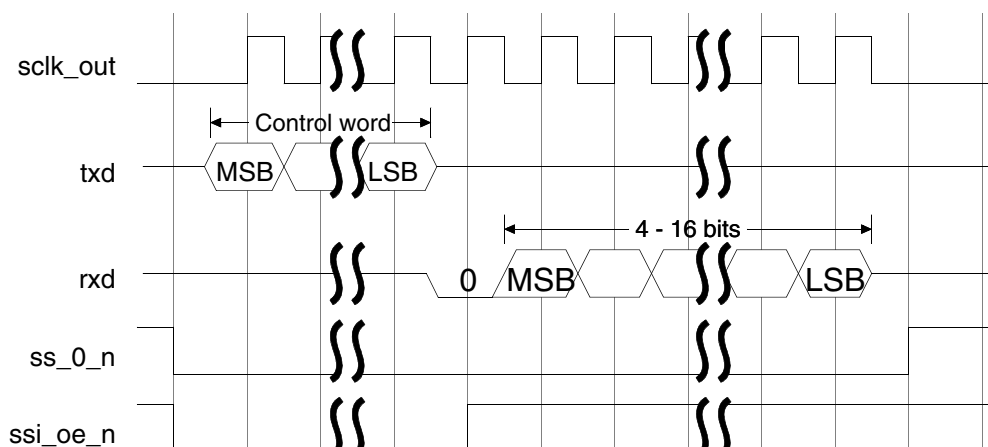
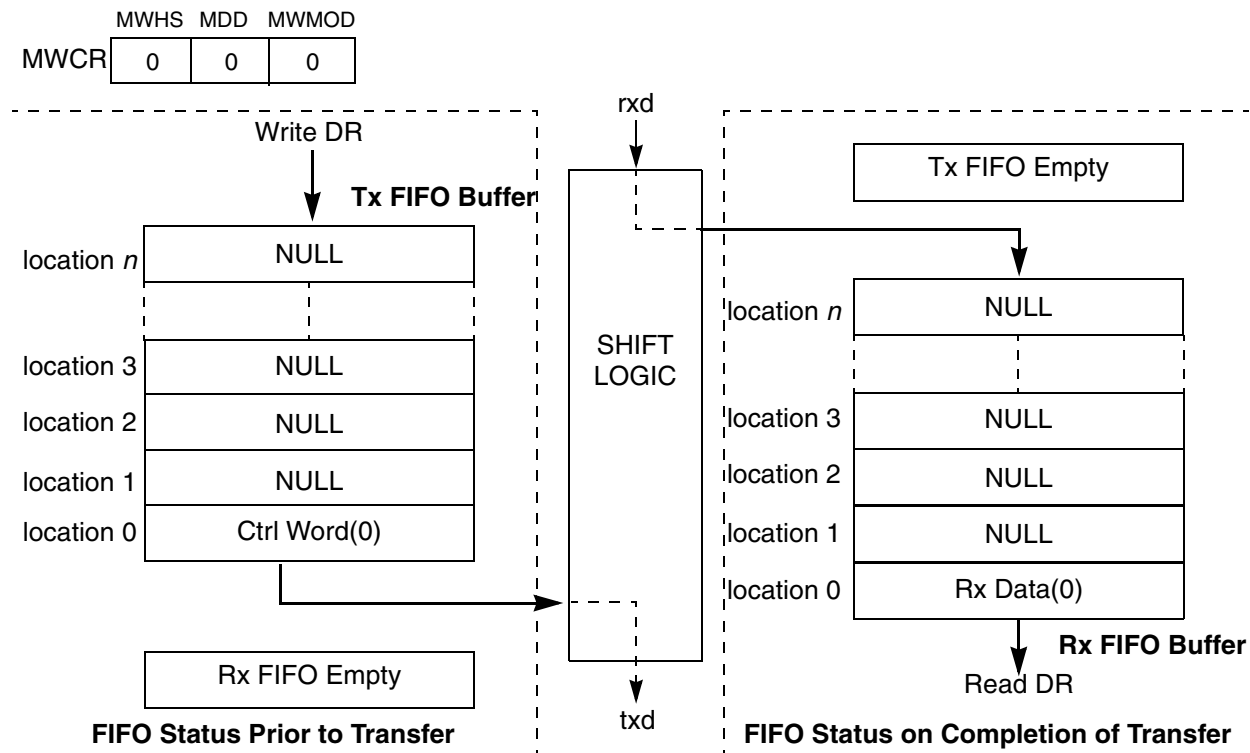


Figure 3-23 shows how the data and control frames are structured in the transmit FIFO prior to the transfer; the value programmed into the [MWCR](#) register is also shown.

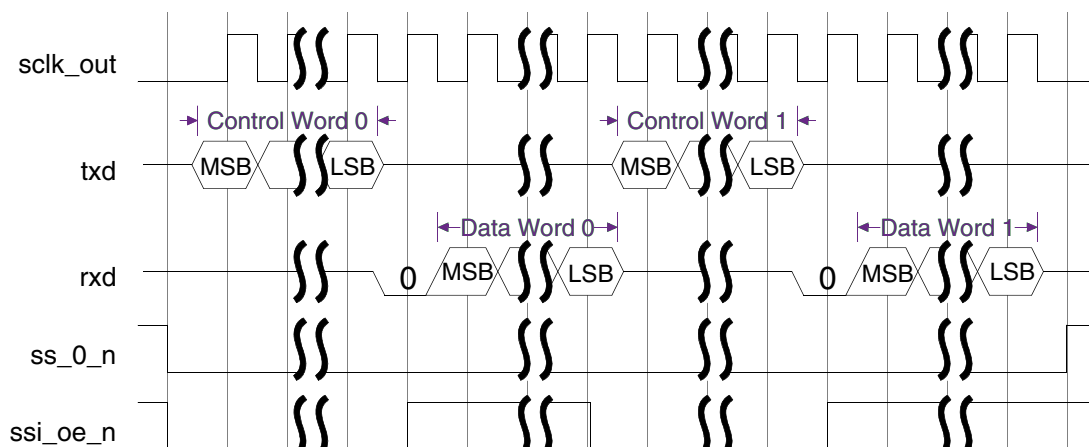
Figure 3-23 FIFO Status for Single Microwire Transfer (receiving data frame)



Continuous transfers from the Microwire protocol can be sequential or nonsequential, and are controlled by the MWMOD bit field (bit 0) in the [MWCR](#) register.

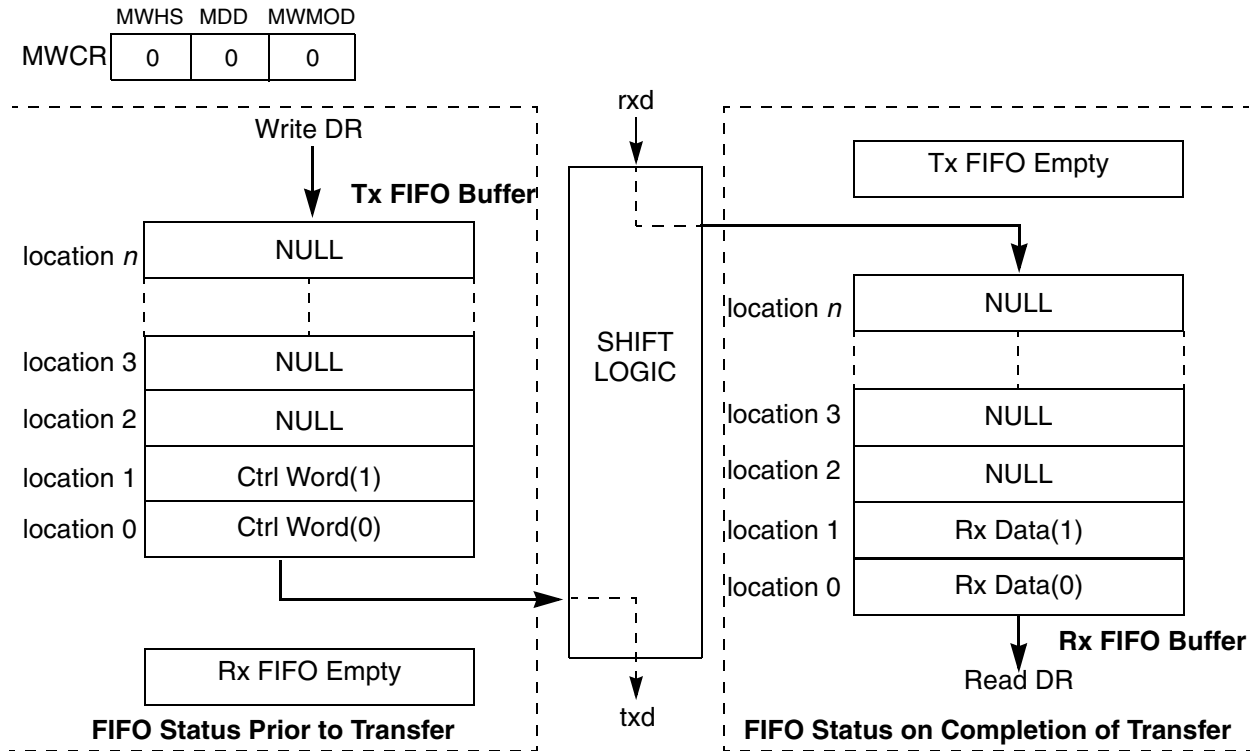
Nonsequential continuous transfers occur as illustrated in [Figure 3-24](#), with the control word for the next transfer following immediately after the LSB of the current data word.

Figure 3-24 Continuous Nonsequential Microwire Transfer (receiving data frame)



The only modification needed to perform a continuous nonsequential transfer is to write more control words into the transmit FIFO buffer; this is illustrated in Figure 3-25. In this example, two data words are read from the external serial-slave device.

Figure 3-25 FIFO Status for Nonsequential Microwire Transfer (receiving data frame)



During sequential continuous transfers, only one control word is transmitted from the DW_apb_ssi master. The transfer is started in the same manner as with nonsequential read operations, but the cycle is continued to read further data. The slave device automatically increments its address pointer to the next location and continues to provide data from that location. Any number of locations can be read in this manner; the DW_apb_ssi master terminates the transfer when the number of words received is equal to the value in the CTRLRI register plus 1.

The timing diagram in [Figure 3-26](#) and example in [Figure 3-27](#) show a continuous sequential read of two data frames from the external slave device.

Figure 3-26 Continuous Sequential Microwire Transfer (receiving data frame)

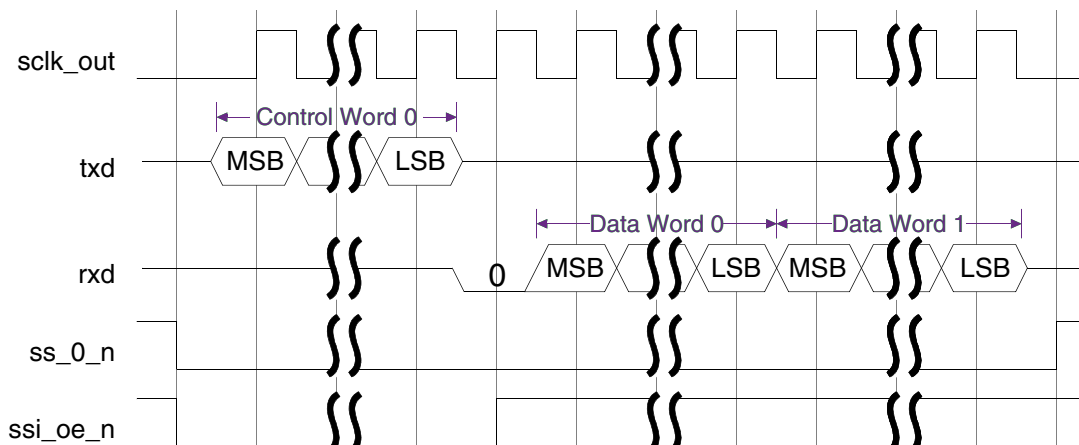
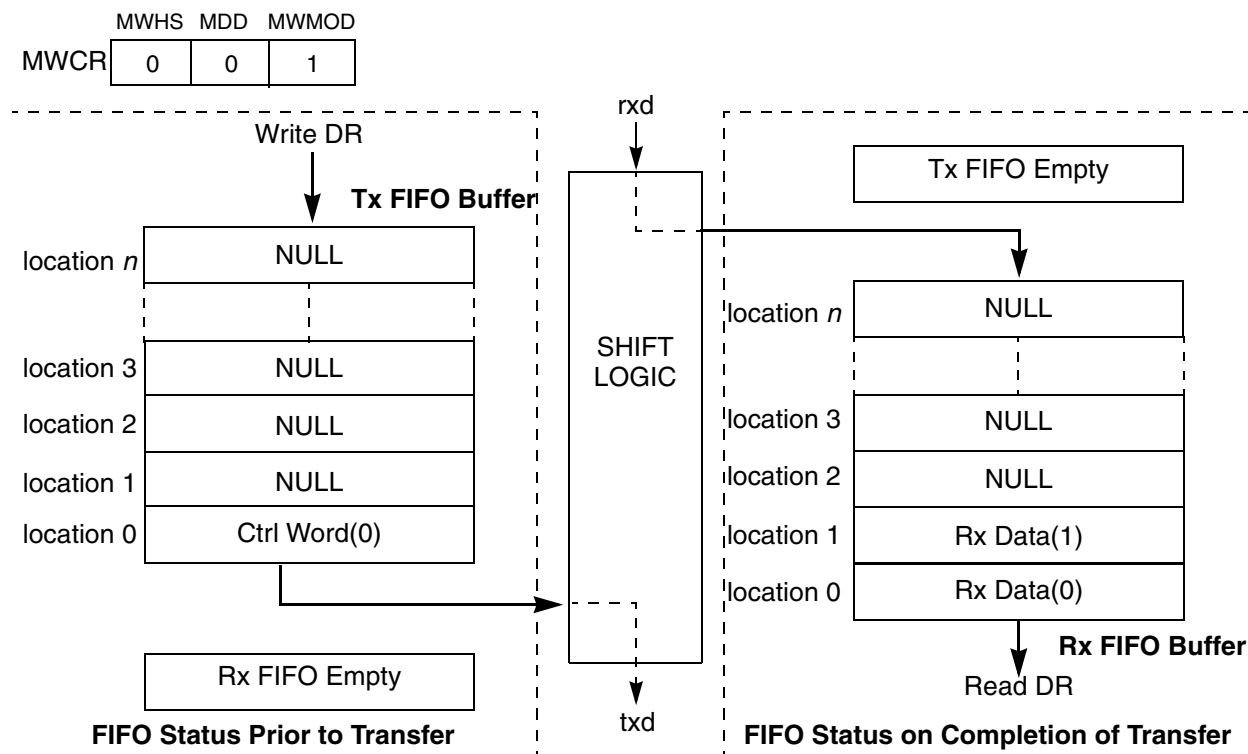


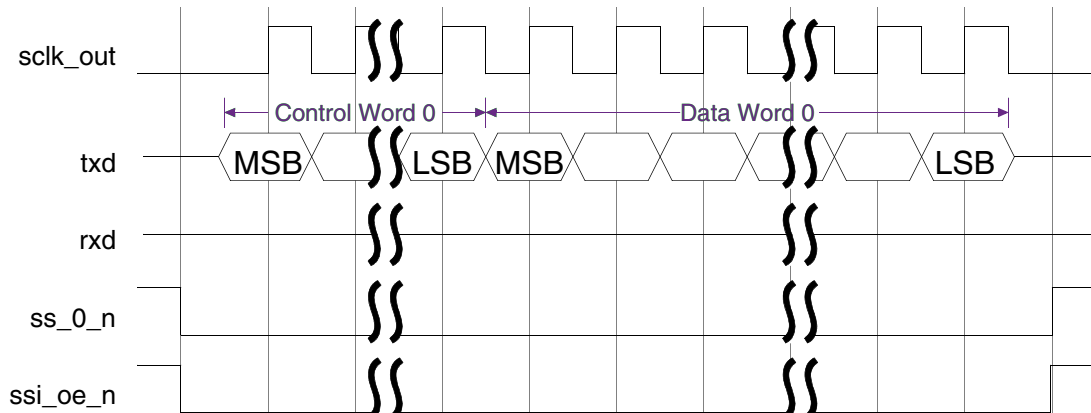
Figure 3-27 FIFO Status for Sequential Microwire Transfer (receiving data frame)



When MDD = 1, this indicates that the DW_apb_ssi serial master transmits data to the external serial slave. Immediately after the LSB of the control word is transmitted, the DW_apb_ssi master begins transmitting

the data frame to the slave peripheral. Figure 3-28 shows the timing diagram for a single DW_apb_ssi serial master write to an external serial slave.

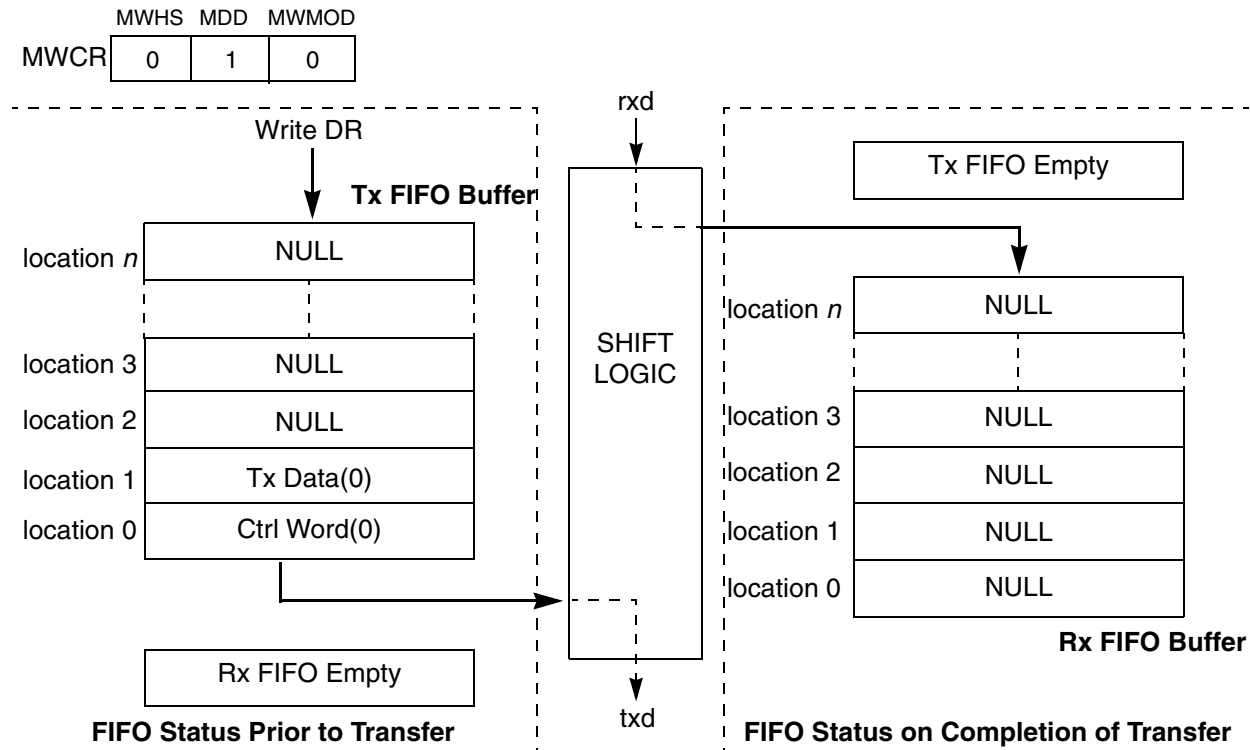
Figure 3-28 Single Microwire Transfer (transmitting data frame)



Note The DW_apb_ssi does not support continuous sequential Microwire writes, where MDD = 1 and MWMOD = 1.

Figure 3-29 shows how the data and control frames are structured in the transmit FIFO prior to the transfer, also shown is the value programmed into the MWCR register.

Figure 3-29 FIFO Status for Single Microwire Transfer (transmitting data frame)



Continuous transfers occur as shown in [Figure 3-30](#), with the control word for the next transfer following immediately after the LSB of the current data word. The only modification you need to make to perform a continuous transfer is to write more control and data words into the transmit FIFO buffer, shown in [Figure 3-31](#). This example shows two data words are written to the external serial slave device.

Figure 3-30 Continuous Microwire Transfer (transmitting data frame)

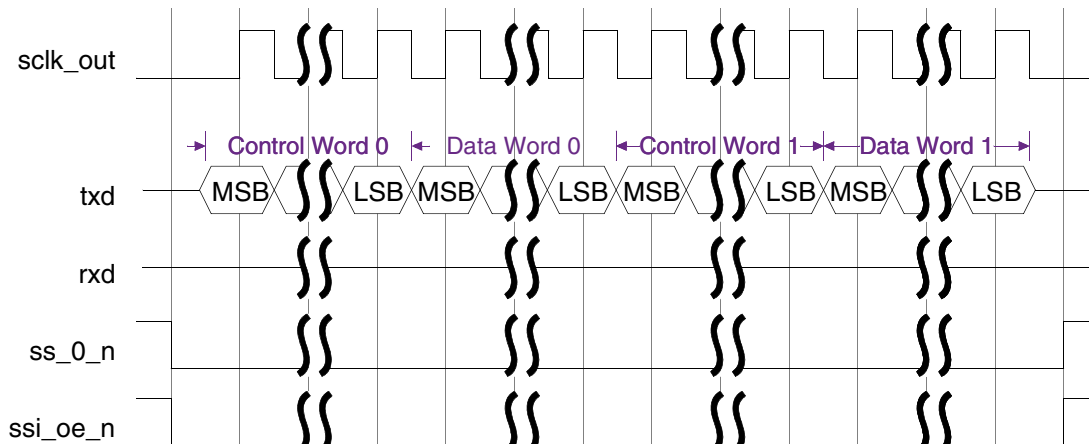
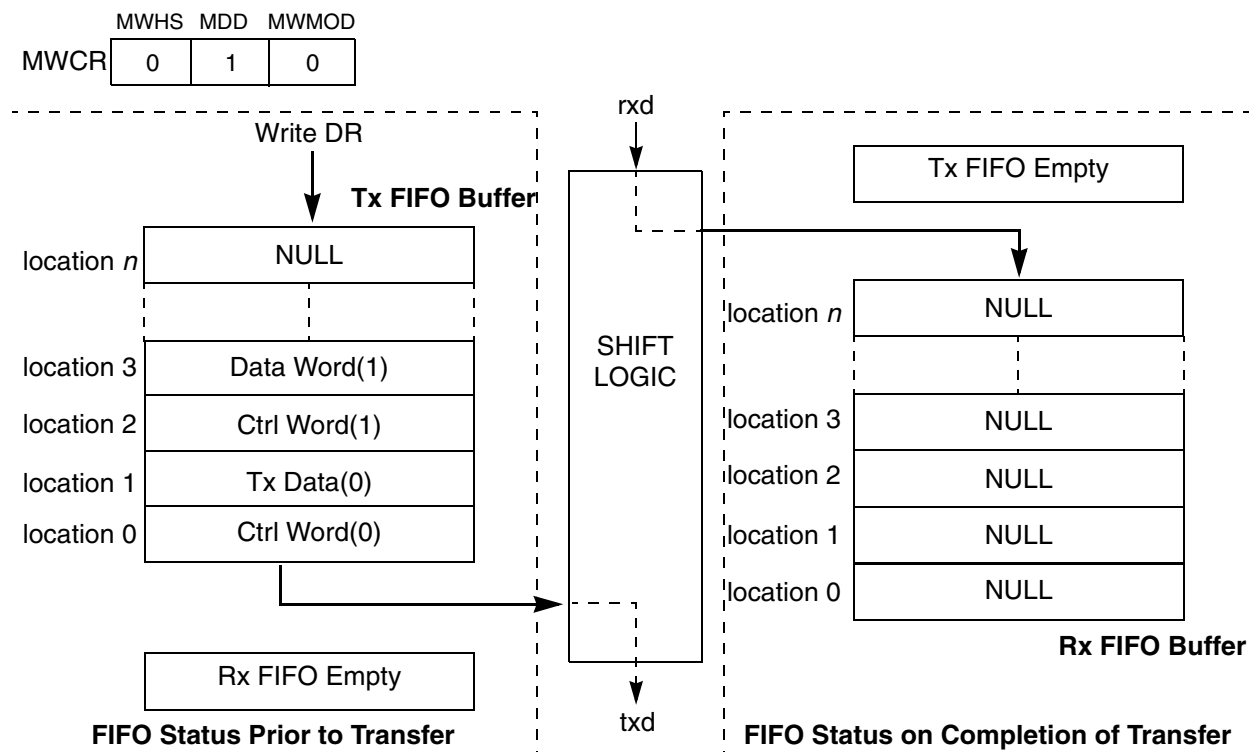


Figure 3-31 FIFO Status for Continuous Microwire Transfer (transmitting data frame)

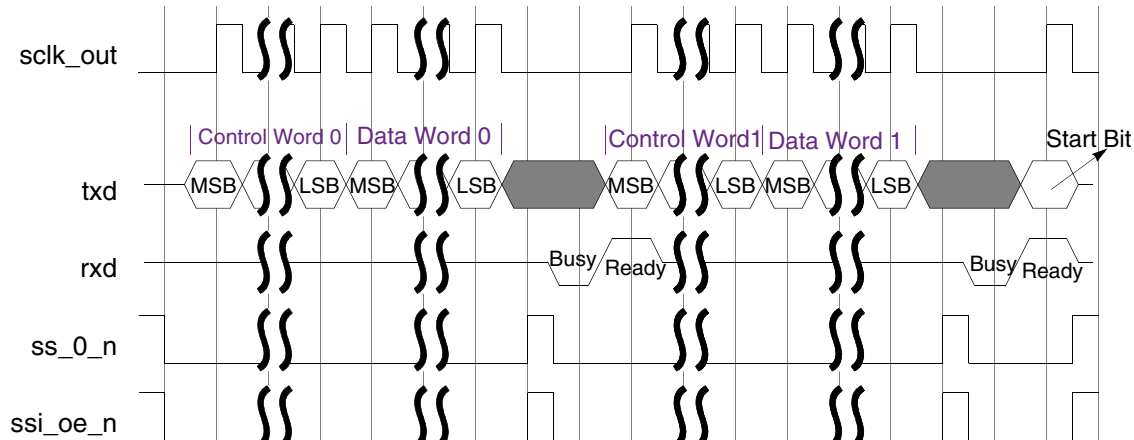


The Microwire handshaking interface can also be enabled for DW_apb_ssi master write operations to external serial-slave devices. To enable the handshaking interface, you must write 1 into the MHS bit field (bit 2) on the [MWCR](#) register. When MHS is set to 1, the DW_apb_ssi serial master checks for a ready status from the slave device before completing the transfer, or transmitting the next control word for continuous

transfers. [Figure 3-32](#) shows an example of a continuous Microwire transfer with the handshaking interface enabled.

After the first data word has been transmitted to the serial-slave device, the DW_apb_ssi master polls the rxd input waiting for a ready status from the slave device. Upon reception of the ready status, the DW_apb_ssi master begins transmission of the next control word. After transmission of the last data frame has completed, the DW_apb_ssi master transmits a start bit to clear the ready status of the slave device before completing the transfer. The FIFO status for this transfer is the same as in [Figure 3-31](#), except that the MWHS bit field is set (1).

Figure 3-32 Continuous Microwire Transfer with Handshaking (transmitting data frame)



To transmit a control word (not followed by data) to a serial-slave device from the DW_apb_ssi master, there must be only one entry in the transmit FIFO buffer. It is impossible to transmit two control words in a continuous transfer, as the shift logic in the DW_apb_ssi treats the second control word as a data word. When the DW_apb_ssi master transmits only a control word, the MDD bit field (bit 1 of [MWCR](#) register) must be set (1).

In the example shown in [Figure 3-33](#) and in the timing diagram in [Figure 3-34](#), the handshaking interface is enabled. If the handshaking interface is disabled (MHS=0), the transfer is terminated by the DW_apb_ssi master one sclk_out cycle after the LSB of the control word is captured by the slave device.

Figure 3-33 FIFO Status for Microwire Control Word Transfer

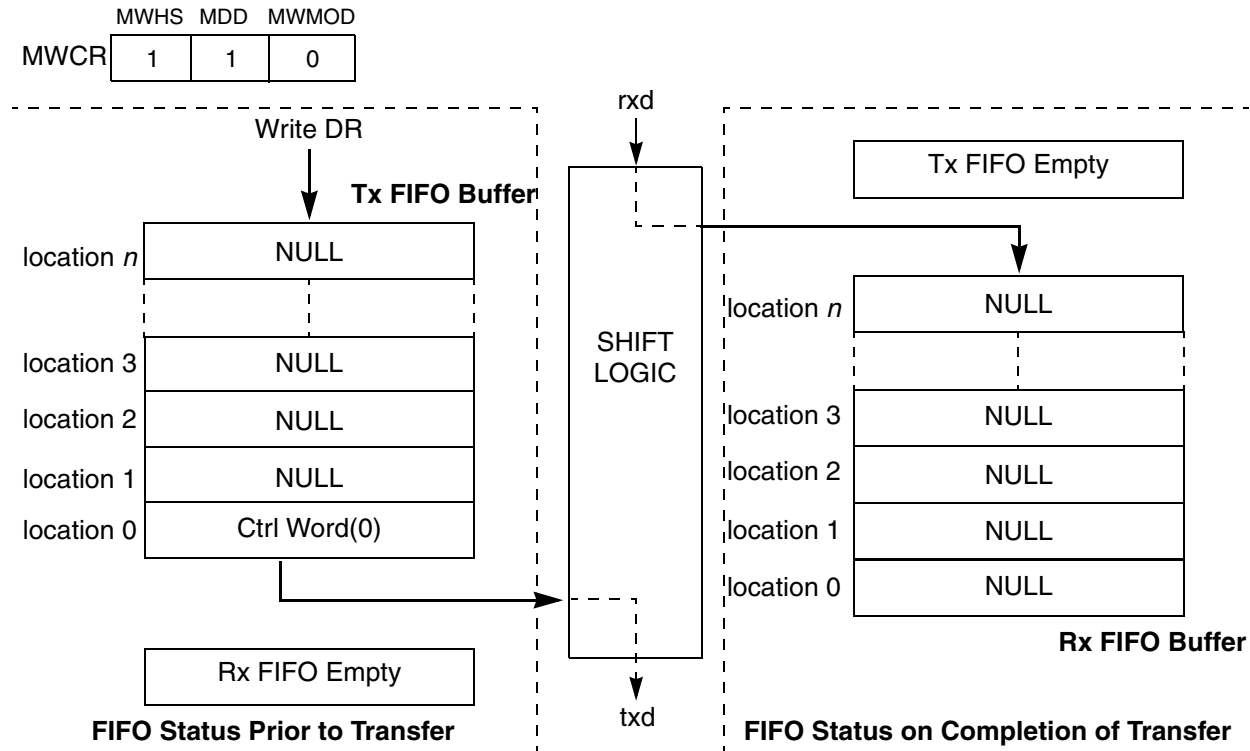
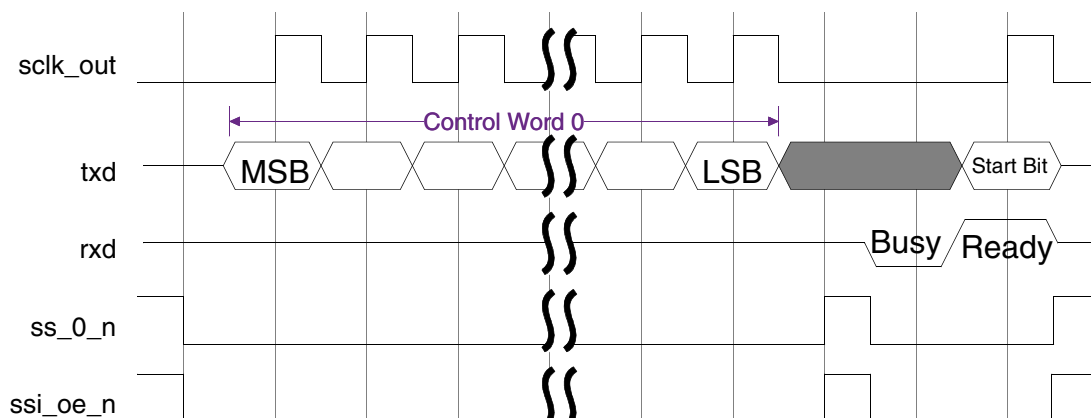


Figure 3-34 Microwire Control Word



When the DW_apb_ssi is configured as a serial slave, data transmission begins with the falling edge of the slave select signal (ss_in_n). One-half serial clock (sclk_in) period later, the first bit of the control is present on the rxd line. The length of the control word can be in the range of 1 to 16 bits and is set by writing bit field CFS in the [CTRLR0](#) register. The CFS bit field must be set to the size of the expected control word from the serial master. The remainder of the control word is received (captured on the rising edge of sclk_in) by the DW_apb_ssi serial slave. During this reception, no data are driven (high impedance) on the serial slave's txd line.

The direction of the data word is controlled by the MDD bit field (bit 1) **MWCR** register. When MDD=0, this indicates that the DW_apb_ssi serial slave is to receive data from the external serial master. Immediately after the control word is transmitted, the serial master begins to drive the data frame onto the DW_apb_ssi slave rxd line. Data are propagated on the falling edge of the serial clock and captured on the rising edge. The slave-select signal is held active-low during the transfer and is de-asserted one-half clock cycle later after the data are transferred. The DW_apb_ssi slave output enable signal (ssi_oe_n) is held inactive for the duration of the transfer. **Figure 3-35** shows the timing diagram for single DW_apb_ssi serial slave read from an external serial master.

Figure 3-35 Single DW_apb_ssi Slave Microwire Serial Transfer (MDD=0)

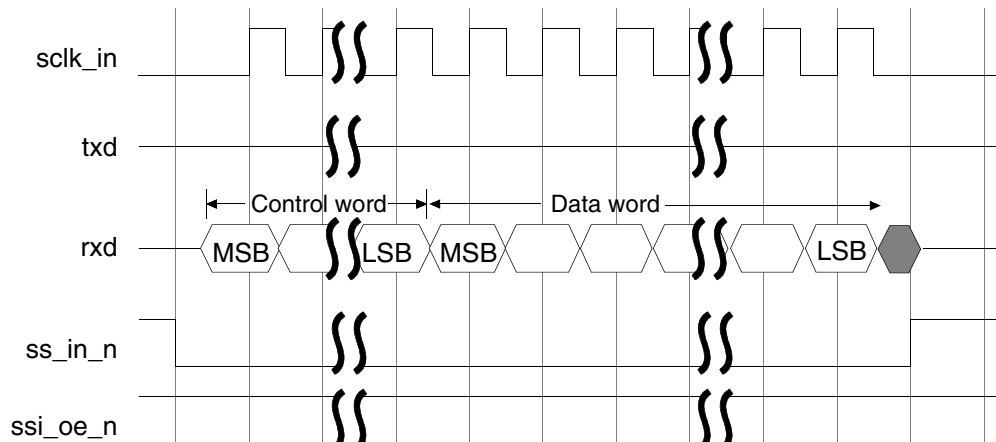
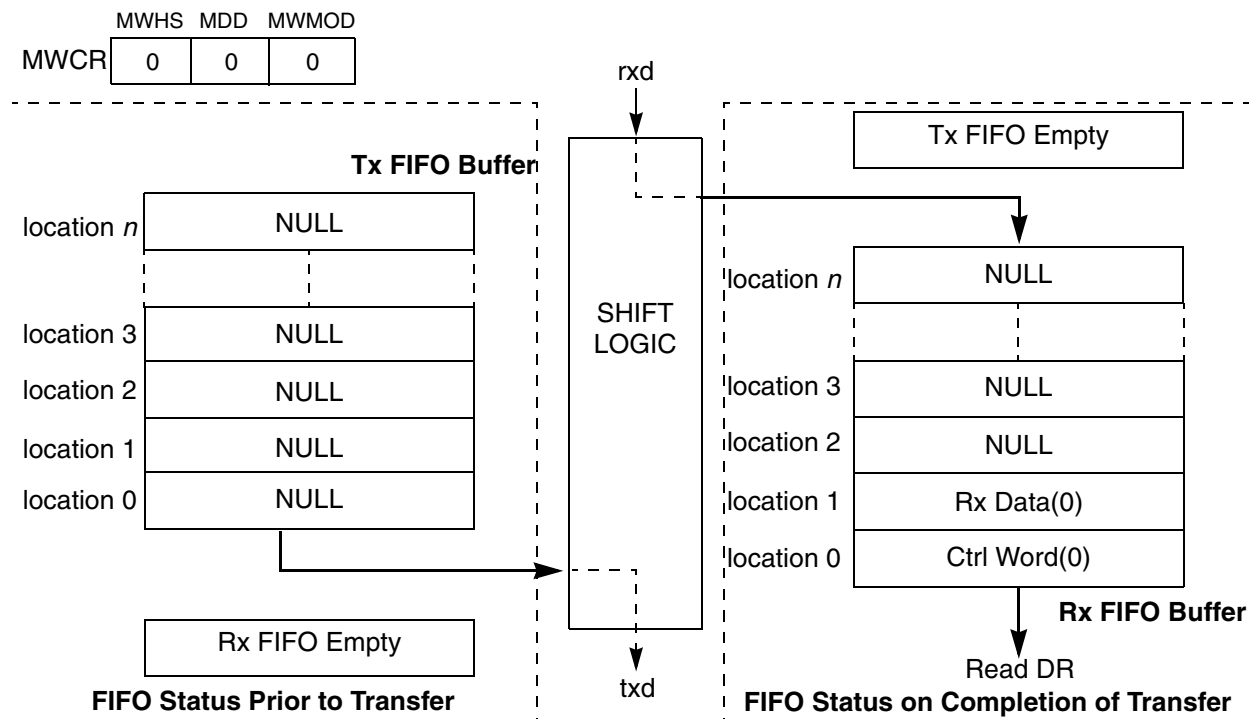


Figure 3-36 shows how the data and control frames are stored in the receive FIFO on completion of the transfer; also shown is the value programmed into the **MWCR** register.

Figure 3-36 FIFO Status for Single Microwire Transfer (receiving data frame)



When MDD=1, this indicates that the DW_apb_ssi serial slave transmits data to the external serial master. Immediately after the LSB of the control word is transmitted, the DW_apb_ssi slave transmits a dummy 0 bit, followed by the 4- to 16-bit data frame on the txd line.

Figure 3-37 shows the timing diagram for a single DW_apb_ssi serial slave write to an external serial master.

Figure 3-37 Single DW_apb_ssi Slave Microwire Serial Transfer (MDD=1)

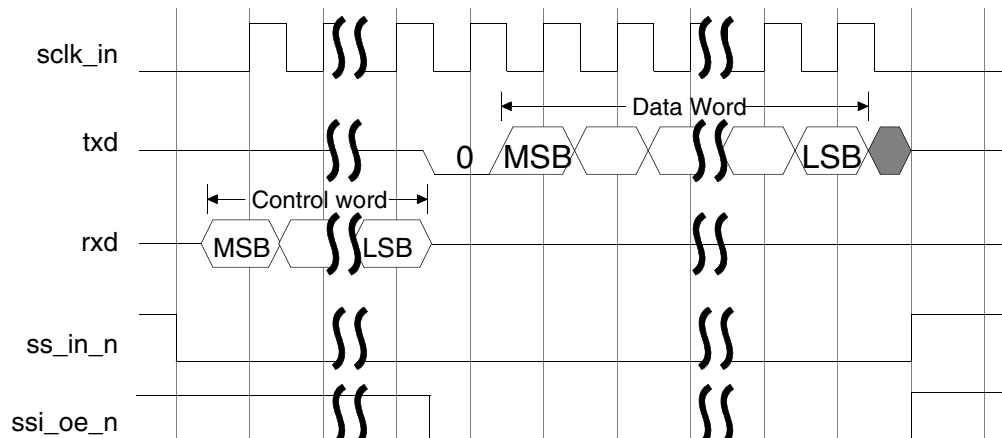
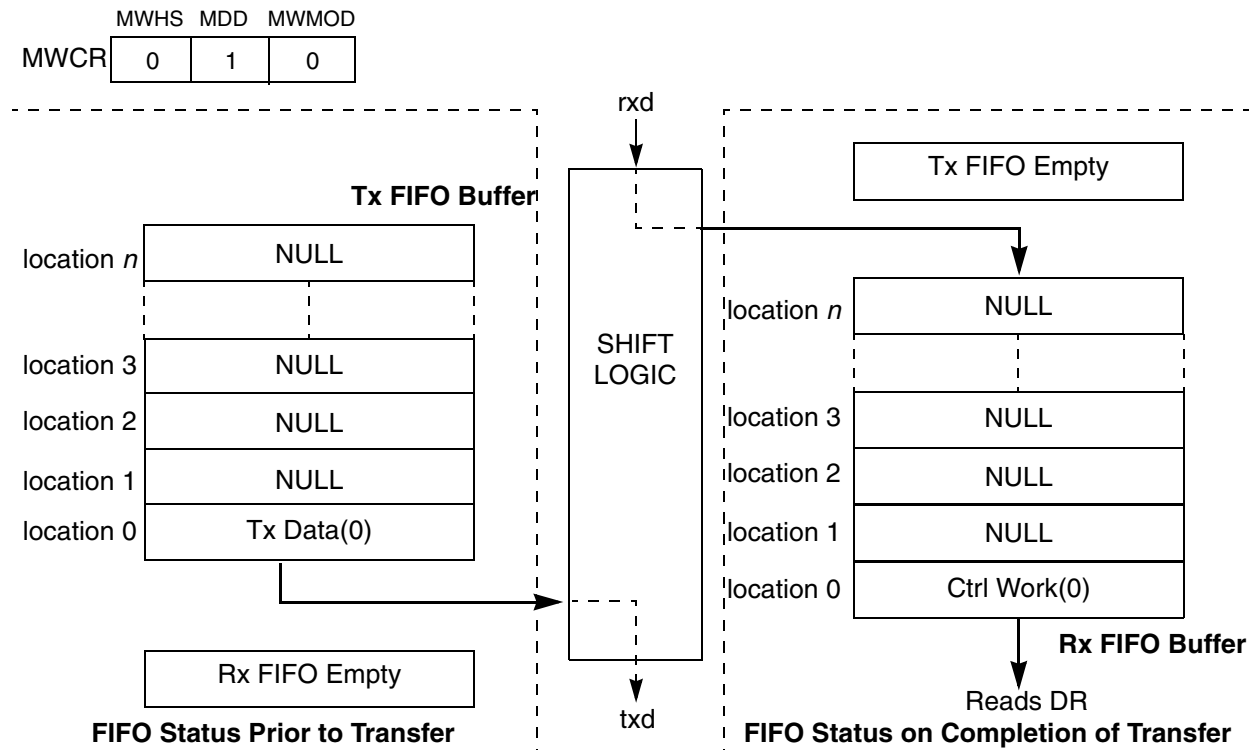


Figure 3-38 FIFO Status for Single Microwire Transfer (transmitting data frame)



Continuous transfers for a DW_apb_ssi slave occur in the same way as those specified for the DW_apb_ssi master configuration. The DW_apb_ssi slave configuration does not support the handshaking interface, as there is never a busy period.

3.5 DMA Controller Interface

The DW_apb_ssi has optional built-in DMA capability which 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_ssi 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_ssi 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_ssi 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 DW_apb_ssi uses two DMA channels, one for the transmit data and one for the receive data. The DW_apb_ssi has these DMA registers:

- ❖ “DMACR” on page 113 – Control register to enable DMA operation.
- ❖ “DMATDLR” on page 114 – Register to set the transmit the FIFO level at which a DMA request is made.
- ❖ “DMARDLR” on page 115 – Register to set the receive FIFO level at which a DMA request is made.

The DW_apb_ssi uses the following handshaking signals to interface with the DMA controller.

- ❖ dma_tx_req
- ❖ dma_tx_single
- ❖ dma_tx_ack
- ❖ dma_rx_req
- ❖ dma_rx_single
- ❖ dma_rx_ack

For more information about these signals, refer to [Table 5-1](#) on page 83. They are discussed further in the “[Handshaking Interface Operation](#)” on page 70.

The DMA output dma_finish is a status signal to indicate that the DMA block transfer is complete; for more information on the dma_finish signal, refer to the Signals chapter in the [DesignWare DW_ahb_dmac Databook](#). The DW_apb_ssi does not use this status signal, and therefore it does not appear in the I/O signal list.

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

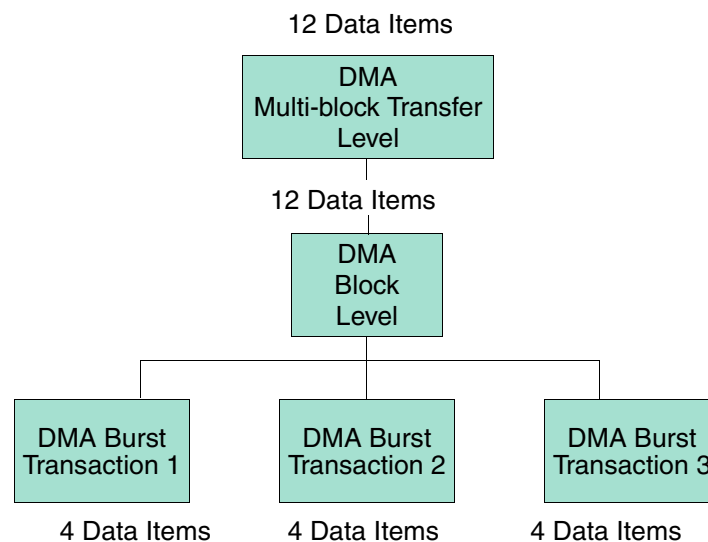
3.5.1 Overview of Operation

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 the DW_apb_ssi; this is programmed into the BLOCK_TS field of the CTLx register.

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

Figure 3-39 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_ssi makes a transmit request to this channel, four data items are written to the DW_apb_ssi transmit FIFO. Similarly, if the DW_apb_ssi makes a receive request to this channel, four data items are read from the DW_apb_ssi receive FIFO. Three separate requests must be made to this DMA channel before all 12 data items are written or read.

Figure 3-39 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_MSIZE = 4

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

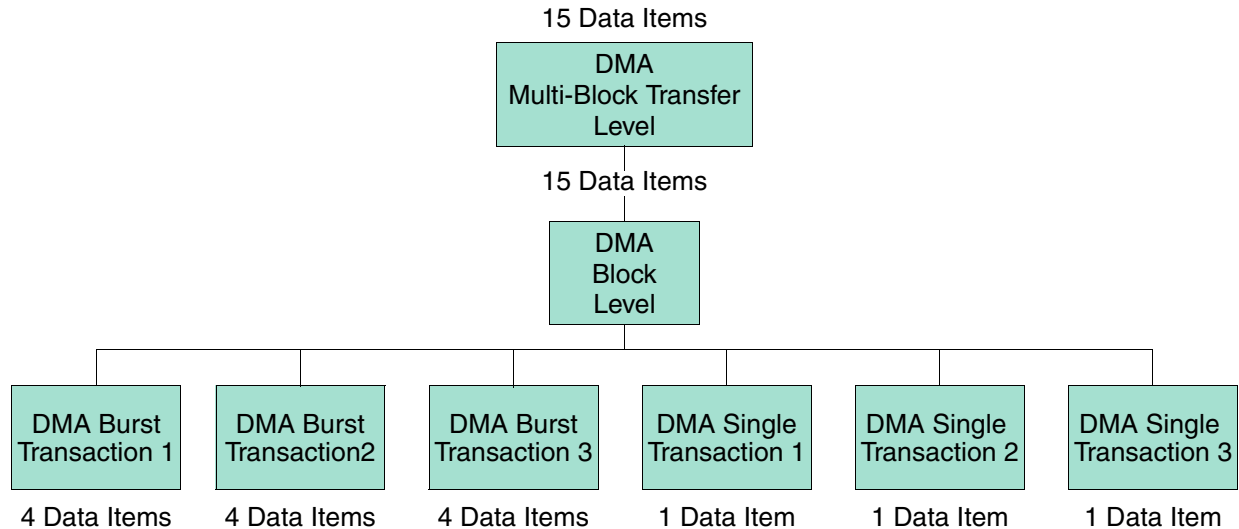


Note

The source and destination transfer width settings in the DW_ahb_dmac – DMA.CTLx.SRC_TR_WIDTH and DMA.CTLx.DEST_TR_WIDTH – should be set to 3'b001 because the DW_apb_ssi FIFOs are 16 bits wide.

When the block size programmed into the DMA Controller is not a multiple of the burst transaction length, as shown in [Figure 3-40](#), a series of burst transactions followed by single transactions are needed to complete the block transfer.

Figure 3-40 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

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

3.5.2 Transmit Watermark Level and Transmit FIFO Underflow

During DW_apb_ssi 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 (**DMATDLR**) 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.

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 (underflow). To prevent this condition, the user must set the watermark level correctly.

3.5.3 Choosing the Transmit Watermark Level

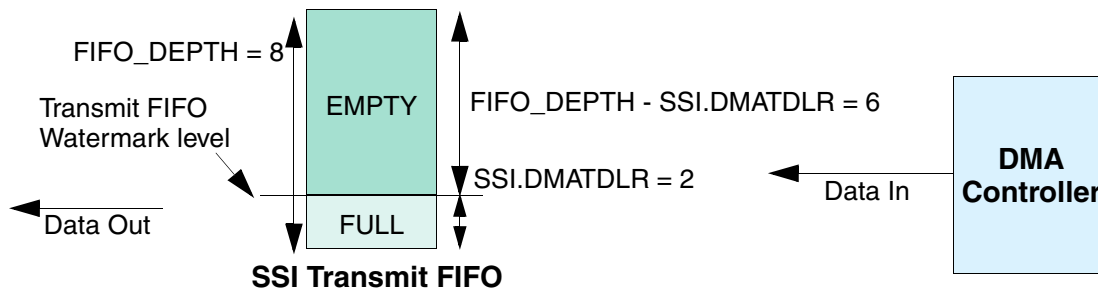
Consider the example where the assumption is made:

$$\text{DMA.CTLx.DEST_MSIZ} = \text{FIFO_DEPTH} - \text{SSI.DMATDLR}$$

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.

3.5.3.0.1 Case 1: DMATDLR = 2

Figure 3-41 Case 1 Watermark Levels



- ❖ Transmit FIFO watermark level = $\text{SSI.DMATDLR} = 2$
- ❖ $\text{DMA.CTLx.DEST_MSIZE} = \text{FIFO_DEPTH} - \text{SSI.DMATDLR} = 6$
- ❖ SSI transmit FIFO_DEPTH = 8
- ❖ $\text{DMA.CTLx.BLOCK_TS} = 30$

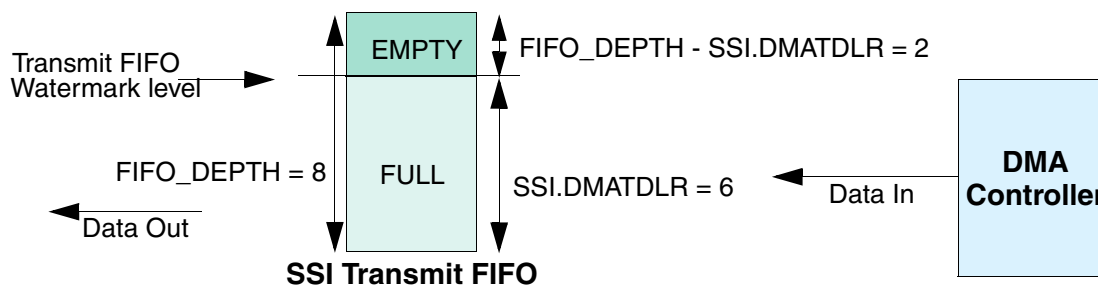
Therefore, the number of burst transactions needed equals the block size divided by the number of data items per burst:

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

The number of burst transactions in the DMA block transfer is 5. But the watermark level, SSI.DMATDLR, is quite low. Therefore, the probability of an SSI underflow is high where the SSI 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.

3.5.3.0.2 Case 2: DMATDLR = 6

Figure 3-42 Case 2 Watermark Levels



- ❖ Transmit FIFO watermark level = $\text{SSI.DMATDLR} = 6$
- ❖ $\text{DMA.CTLx.DEST_MSIZE} = \text{FIFO_DEPTH} - \text{SSI.DMATDLR} = 2$
- ❖ SSI transmit FIFO_DEPTH = 8
- ❖ $\text{DMA.CTLx.BLOCK_TS} = 30$

Number of burst transactions in Block:

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

In this block transfer, there are 15 destination burst transactions in a DMA block transfer. But the watermark level, SSI.DMATDLR, is high. Therefore, the probability of an SSI underflow is low because the DMA

controller has plenty of time to service the destination burst transaction request before the SSI 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 SSI 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.

3.5.4 Selecting DEST_MSIZ and Transmit FIFO Overflow

As can be seen from [Figure 3-42](#) on page 68, 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 SSI transmit FIFO to service the destination burst request. Therefore, the following equation must be adhered to in order to avoid overflow:

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

In [Case 2: DMATDLR = 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:

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

This is the setting used in [Figure 3-40](#) on page 67.

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



Note

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

3.5.5 Receive Watermark Level and Receive FIFO Overflow

During DW_apb_ssi 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, DMARDLR+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.

3.5.6 Choosing the Receive Watermark level

Similar to choosing the transmit watermark level described earlier, the receive watermark level, $DMARDLR+1$, should be set to minimize the probability of overflow, as shown in Figure 3-43. It is a tradeoff between the number of DMA burst transactions required per block versus the probability of an overflow occurring.

3.5.7 Selecting SRC_MSIZ and Receive FIFO Underflow

As can be seen in Figure 3-43, 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 – $DMA.CTLx.SRC_MSIZE$ – the receive FIFO may be emptied, but not underflowed, at the completion of the burst transaction. For optimal operation, $DMA.CTLx.SRC_MSIZE$ should be set at the watermark level; that is:

$$DMA.CTLx.SRC_MSIZE = SSI.DMARDLR + 1 \quad (3)$$

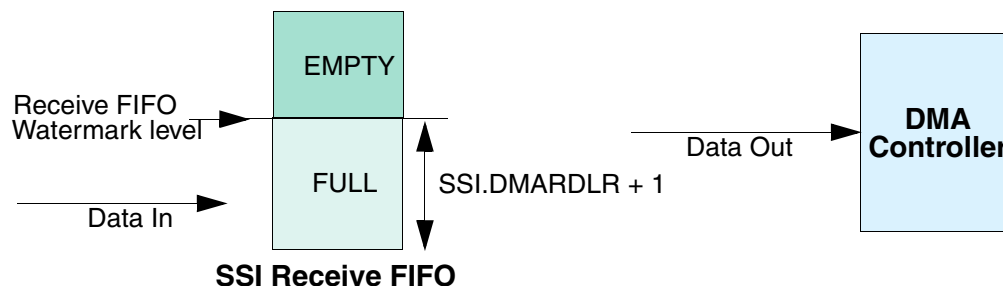
Adhering to equation (3) reduces the number of DMA bursts in a block transfer, and this in turn can improve AMBA bus utilization.



Note

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

Figure 3-43 SSI Receive FIFO



3.5.8 Handshaking Interface Operation

The following sections discuss the DW_apb_ssi handshaking interface.

3.5.8.1 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_ssi 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_ssi 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 3-44 on page 71 shows a timing diagram of a burst transaction where $pclk = hclk$. Figure 3-45 shows two back-to-back burst transactions where the $hclk$ frequency is twice the $pclk$ frequency.

The handshaking loop is as follows :

- dma_tx_req/dma_rx_req asserted by DW_apb_ssi
- > dma_tx_ack/dma_rx_ack asserted by DW_ahb_dmac
- > dma_tx_req/dma_rx_req de-asserted by DW_apb_ssi
- > dma_tx_ack/dma_rx_ack de-asserted by DW_ahb_dmac
- > dma_tx_req/dma_rx_req re-asserted by DW_apb_ssi, if back-to-back transaction is required



Note

The burst transaction request signals, `dma_tx_req` and `dma_rx_req`, are generated in the DW_apb_ssi 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_ssi of `pclk`. The handshaking mechanism between the DW_ahb_dmac and the DW_apb_ssi supports quasi-synchronous clocks; that is, `hclk` and `pclk` must be phase-aligned, and the `hclk` frequency must be a multiple of the `pclk` frequency.

Figure 3-44 Burst Transaction – $pclk = hclk$

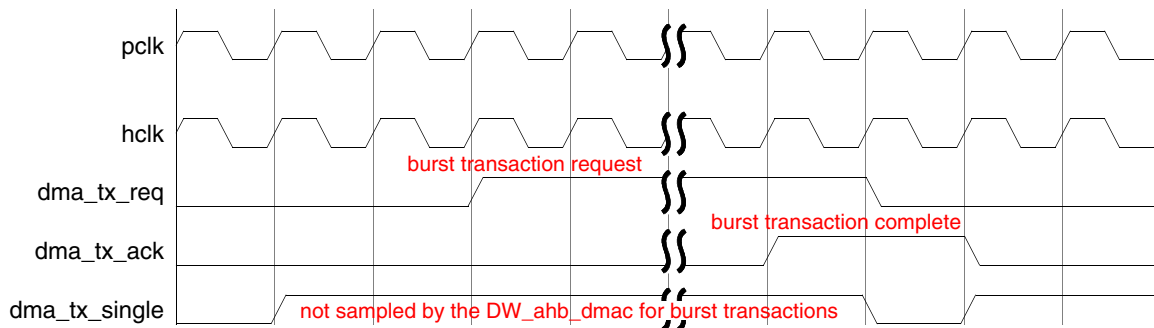
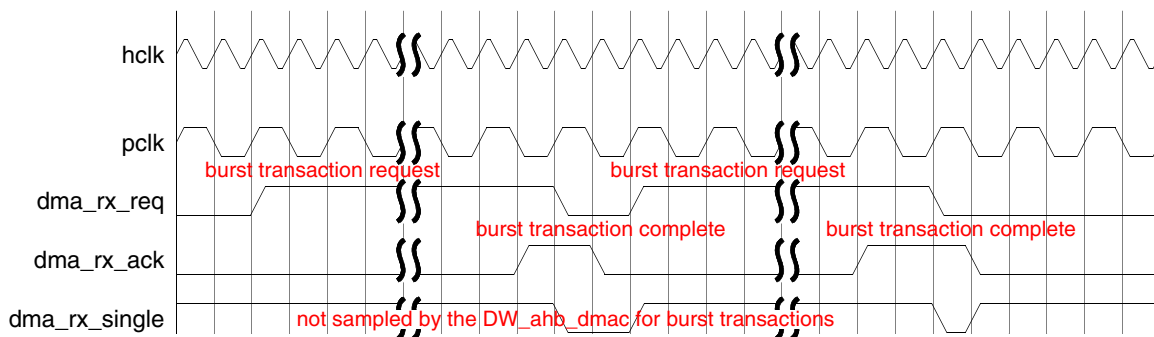


Figure 3-45 Back-to-Back Burst Transactions – $hclk = 2 \cdot pclk$



Two things to note here:

1. The burst request lines, `dma_tx_req` signal/ `dma_rx_req`, once asserted remain asserted until their corresponding `dma_tx_ack`/ `dma_rx_ack` signal is received even if the respective FIFO's drop below their watermark levels during the burst transaction.

2. The dma_tx_req/dma_rx_req signals are de-asserted when their corresponding dma_tx_ack/dma_rx_ack signals are asserted, even if the respective FIFOs exceed their watermark levels.

3.5.8.2 dma_tx_single, dma_rx_single

The dma_tx_single signal is asserted when there is at least one free entry in the transmit FIFO, and is cleared when the transmit FIFO is full or the dma_tx_ack signal is active. The dma_tx_single signal will be re-asserted when the dma_tx_ack signal is removed, if the condition for setting still holds true.

The dma_rx_single signal is asserted when there is at least one valid data entry in the receive FIFO, and is cleared when the receive FIFO is empty or the dma_rx_ack signal is active. The dma_rx_single signal will be re-asserted when the dma_rx_ack signal is removed, if the condition for setting still holds true.

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_MSIZ, CTLx.DEST_MSIZ, as shown in [Figure 3-40](#) on page 67. In this case, the DMA single outputs inform the DW_ahb_dmac that it is still possible to perform single data item transfers, so it can access all data items in the transmit/receive FIFO and complete the DMA block transfer. The DMA single outputs from the DW_apb_ssi 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_ssi is as follows:

DMA.CTLx.SRC_MSIZ = SSI.DMARDLR + 1 = 4

DMA.CTLx.BLOCK_TS = 12

For the example in [Figure 3-39](#), 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_ssi serial transfer, ensuring that all 12 data items are read by the DW_ahb_dmac. All DMA requests read a block of data items and no single DMA transactions are required. This block transfer is made up of three burst transactions.

Now, for the following block transfer:

DMA.CTLx.SRC_MSIZ = SSI.DMARDLR + 1 = 4

DMA.CTLx.BLOCK_TS = 15

The first 12 data items are transferred as already described using 3 burst transactions. But when the last three data frames enter the receive FIFO, the dma_rx_req 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 3-46](#) shows a single transaction. The handshaking loop is as follows:

```

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

```

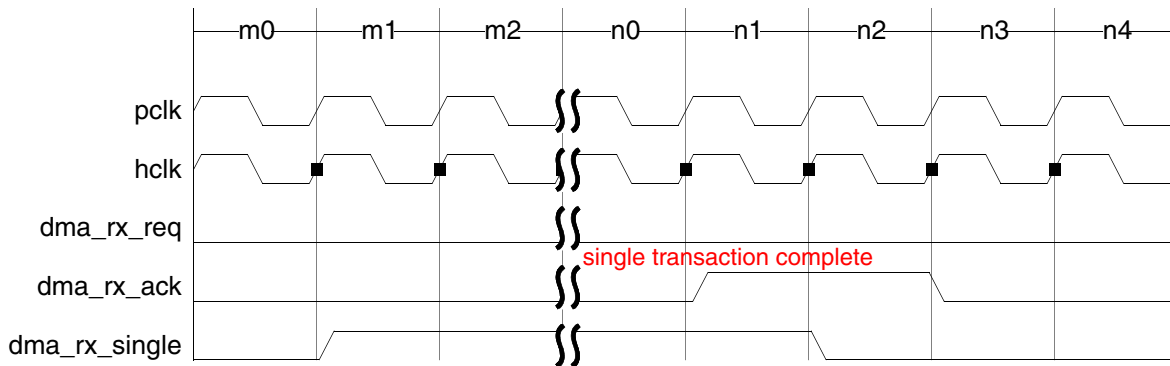
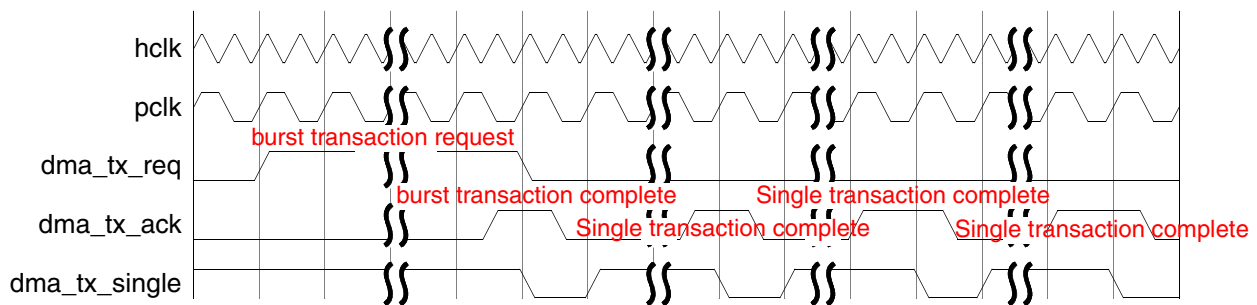

Figure 3-46 Single Transaction

Figure 3-47 shows a burst transaction, followed by three back-to-back single transactions, where the hclk frequency is twice the pclk frequency.

Figure 3-47 Burst Transaction + 3 Back-to-Back Singles – $hclk = 2 \times pclk$ **Note**

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

3.6 APB Interface

The host processor accesses data, control, and status information on the DW_apb_ssi through the APB interface. The DW_apb_ssi supports APB data bus widths of 8, 16, and 32 bits. APB accesses to the DW_apb_ssi peripheral are described in the following subsections.

3.6.1 Control and Status Register APB Access

Control and status registers within the DW_apb_ssi are byte-addressable. The maximum width of the control or status register in the DW_apb_ssi is 16 bits. Therefore, if the APB data bus is 16 or 32 bits wide, all read and write operations to the DW_apb_ssi control and status registers require only one APB access. When the APB data bus width is 8 bits, you may independently write to the lower byte lane [7:0] of a register by accessing the base address of the register. The upper byte lane [15:8] can be accessed by addressing the register base address + 1.

3.6.2 Data Register APB Access

The data register (DR) within the DW_apb_ssi is 16 bits wide in order to remain consistent with the maximum serial transfer size (data frame). An APB write operation to DR moves data from pwwdata into the transmit FIFO buffer. An APB read operation from DR moves data from the receive FIFO buffer onto prdata.

When the APB data bus is 8 bits wide, you must perform two APB accesses to write or read to and from this register. When data are written to the DW_apb_ssi DR, the lower byte [7:0] is stored in an intermediate 8-bit register. Only when the second APB write occurs are the 16 bits of data loaded into the DW_apb_ssi transmit FIFO buffer. When data are read from the DW_apb_ssi DR, the upper byte [15:8] is stored in an intermediate 8-bit register. The lower byte [7:0] is returned on the first APB read, and the stored upper byte [15:8] is returned on the second APB read.

When the APB data bus is 16 bits wide, the DW_apb_ssi DR can be written or read in one APB access. This is the optimal configuration because the full bandwidth of the bus is utilized when accessing this peripheral. When the APB data bus is 32 bits wide, the DW_apb_ssi DR can be written or read in one APB access. It is impossible to write or read two 16-bit FIFO entries in a single 32-bit APB access. Therefore, only half of the APB bus bandwidth is utilized when accessing DW_apb_ssi from a 32-bit APB bus.

**Note**


The **DR** register in the DW_apb_ssi occupies sixty-four 32-bit locations of the memory map to facilitate AHB burst transfers. There are no burst transactions on the APB bus itself, but DW_apb_ssi supports the AHB bursts that happen on the AHB side of the AHB/APB bridge. Writing to any of these address locations has the same effect as pushing the data from the pwwdata bus into the transmit FIFO. Reading from any of these locations has the same effect as popping data from the receive FIFO onto the prdata bus. The FIFO buffers on the DW_apb_ssi are not addressable.

For more information about the APB Interface and data widths, refer to “[Integration Considerations](#)” on page [127](#).

4

Parameters


This chapter describes the parameters used by the DW_apb_ssi. You use coreConsultant or coreAssembler to configure the following parameters and generate the configured code.

 **Attention**

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

4.1 Parameter Descriptions

Table 4-1 lists the DW_apb_ssi parameter descriptions.

 **Note**

In Table 4-1, 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 as check boxes and drop-down lists in the coreConsultant GUI.

Table 4-1 Top-Level Parameters

Field Label	Parameter Definition
System Configuration	
APB Data Bus Width	Parameter Name: APB_DATA_WIDTH Legal Values: 8, 16, or 32 Default Value: 32 Dependencies: None. Description: Width of the APB data bus
Device Configuration	
Serial master or slave configuration	Parameter Name: SSI_IS_MASTER Legal Values: Slave (0) or Master (1) Default Value: Master (1) Dependencies: None. Description: Configures the device as a master or slave serial peripheral.

Table 4-1 Top-Level Parameters (Continued)

Field Label	Parameter Definition
Receive FIFO buffer depth	Parameter Name: SSI_RX_FIFO_DEPTH Legal Values: 2 to 256 Default Value: 8 Dependencies: None. Description: Configures the depth of the receive FIFO buffer.
Transmit FIFO buffer depth	Parameter Name: SSI_TX_FIFO_DEPTH Legal Values: 2 to 256 Default Value: 8 Dependencies: None. Description: Configures the depth of the transmit FIFO buffer.
Number of slave select lines	Parameter Name: SSI_NUM_SLAVES Legal Values: 1 to 16 Default Value: 1 Dependencies: Parameter is valid only when SSI_IS_MASTER is set to Master (1). Description: Configures the number of slave select lines from the DW_apb_ssi master.
Peripheral ID Code	Parameter Name: SSI_ID Legal Values: 0x00000000 to 0xFFFFFFFF Default Value: 0xffffffff Dependencies: None. Description: Individual peripheral identification code.
Include Programmable RXD Sample Logic	Parameter Name: SSI_HAS_RX_SAMPLE_DELAY Legal Values: True (1) or False (0) Default Value: False (0) Dependencies: Parameter is valid only when SSI_IS_MASTER is set to Master (1). Description: Includes logic to allow a programmable delay on the sample time of the rxd input signal. When this logic is included, the default sample time of the rxd signal can be delayed by a programmable number of ssi_clk cycles.
Maximum RXD Sample Delay	Parameter Name: SSI_RX_DLY_SR_DEPTH Legal Values: 4 to 255 Default Value: 4 Dependencies: Parameter is valid only when SSI_HAS_RX_SAMPLE_DELAY is set to 1. Description: Defines the maximum number of ssi_clk cycles that can be used to delay the sampling of the rxd input. Two registers are added to design logic for each value.

Table 4-1 Top-Level Parameters (Continued)

Field Label	Parameter Definition
External Configuration	
Include DMA Handshaking Interface Signals?	<p>Parameter Name: SSI_HAS_DMA</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: False (0)</p> <p>Dependencies: None.</p> <p>Description: When configured, includes the DMA interface signals at the top-level I/O.</p>
Configure interrupt pin out	<p>Parameter Name: SSI_INTR_IO</p> <p>Legal Values: Individual Interrupts (0) or Combined Interrupt (1)</p> <p>Default Value: Individual Interrupts (0)</p> <p>Dependencies: None.</p> <p>Description: Selects which interrupt-related signals appear as outputs of the design. You have a choice of having either a single combined interrupt (the logical OR of all DW_apb_ssi interrupt outputs) or have each individual interrupt appear as a separate output signal on the component. When configured as a master, there are six individual interrupts. When configured as a slave, there are five individual interrupts.</p>
Active interrupt level	<p>Parameter Name: SSI_INTR_POL</p> <p>Legal Values: Active-low or active-high (0 or 1)</p> <p>Default Value: Active-low (0)</p> <p>Dependencies: None.</p> <p>Description: Configures the active level of the output interrupt lines.</p>
Are pclk and ssi_clk synchronous?	<p>Parameter Name: SSI_SYNC_CLK</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: True (1)</p> <p>Dependencies: None.</p> <p>Description: Defines if the pclk is synchronous to the ssi_clk. If they are synchronous, then you do not need to re-time signals across the clock domains.</p>
Generate clock enable input for ssi_clk?	<p>Parameter Name: SSI_CLK_EN_MODE</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: False (0)</p> <p>Dependencies: None.</p> <p>Description: When enabled, the ssi_clk_en signal enables data propagation through ssi_clk flip-flops. When disabled, the ssi_clk flip-flops are always enabled.</p>

Table 4-1 Top-Level Parameters (Continued)

Field Label	Parameter Definition
Internal Configuration	
Hard code the frame format?	<p>Parameter Name: SSI_HC_FRF</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: False (0)</p> <p>Dependencies: None.</p> <p>Description: When set, the frame format (serial protocol) can be fixed so that you cannot change it. This setting restricts the use of DW_apb_ssi to be only a single-frame format peripheral.</p>
Default frame format	<p>Parameter Name: SSI_DFLT_FRF</p> <p>Legal Values: Motorola SPI, TI SSP, NatSemi MicroWire (0x0, 0x1, 0x2)</p> <p>Default Value: Motorola SPI (0x0)</p> <p>Dependencies: None.</p> <p>Description: Selects the frame format that is available directly after reset. You can select any of the formats to be the default. If the frame format is hardcoded, the default frame format is the only one possible.</p>
Default serial clock polarity	<p>Parameter Name: SSI_DFLT_SCPOL</p> <p>Legal Values: 0 or 1</p> <p>Default Value: 0</p> <p>Dependencies: Used only when the frame format is Motorola SPI (SSI_DFLT_FRF = Motorola SPI).</p> <p>Description: Defines the SPI serial clock polarity available directly after reset. When set to 0, the inactive level of the SPI serial clock is 0. When set to 1, the inactive level of the SPI serial clock is 1.</p>
Default serial clock phase	<p>Parameter Name: SSI_DFLT_SCPH</p> <p>Legal Values: 0 or 1</p> <p>Default Value: 0</p> <p>Dependencies: SSI_DFLT_FRF = Motorola SPI</p> <p>Description: Defines the SPI serial clock phase available directly after reset. When set to 0, SPI serial data are captured on the first edge of the serial clock. When set to 1, SPI serial data are captured on the second edge of the serial clock.</p>

Table 4-2 lists the derived configuration parameters.

Table 4-2 Derived Configuration Parameters

Parameter	Description
TX_ABW	<p>Transmit FIFO address bus width. This value is equal to the integer result of:</p> <p>$\log_2(\text{SSI_TX_FIFO_DEPTH})$</p> <p>Range: 1 to 8</p>
RX_ABW	<p>Receive FIFO address bus width. This value is equal to the integer result of:</p> <p>$\log_2(\text{SSI_RX_FIFO_DEPTH})$</p> <p>Range: 1 to 8</p>

5

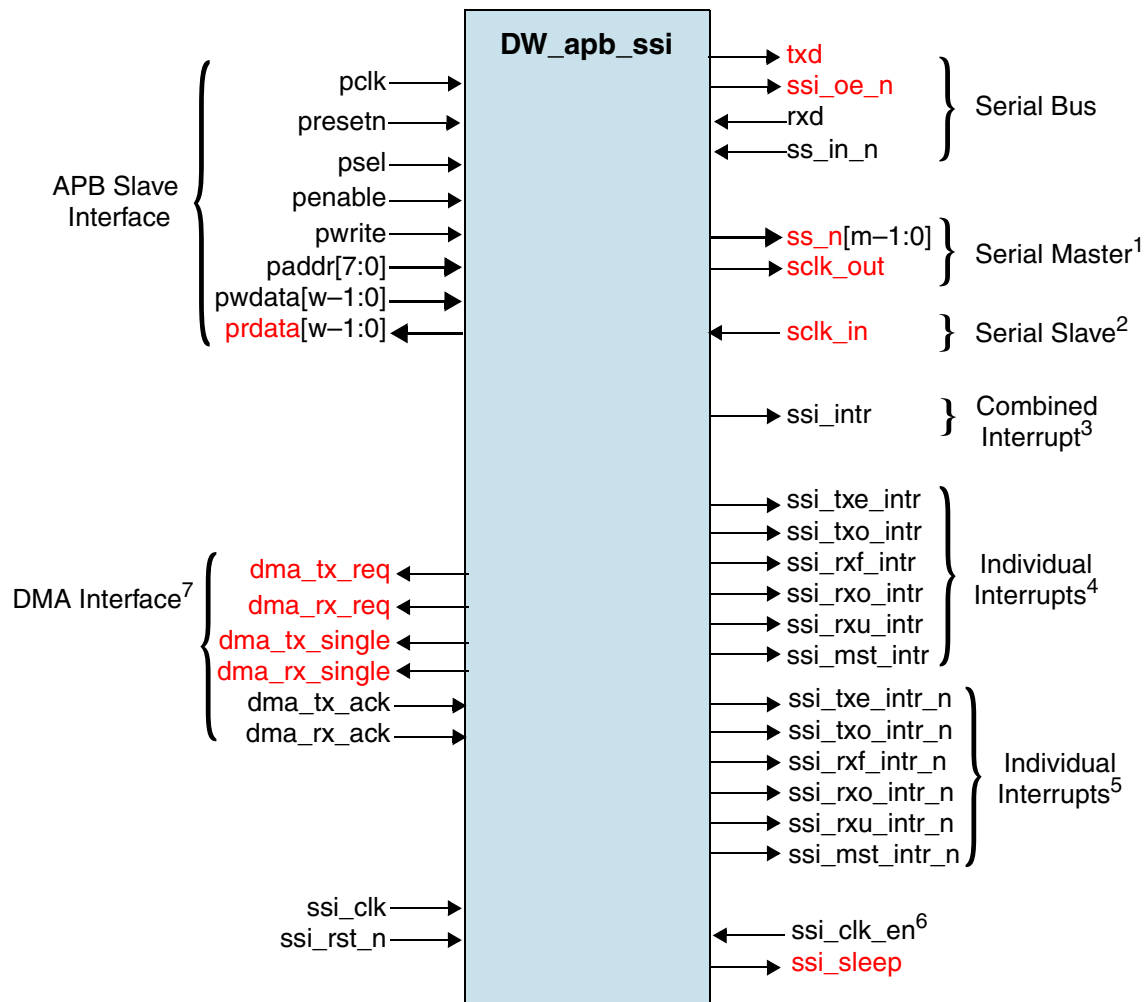
Signals

The following subsections describe how to interface to the DW_apb_ssi.

5.1 DW_apb_ssi Interface Diagram

[Figure 5-1](#) shows the I/O signals for DW_apb_ssi (refer to [Table 5-1](#) on page 83).

Figure 5-1 DW_apb_ssi Interface Diagram



w = APB data bus width (APB_DATA_WIDTH)

m = Number of slave devices on the serial bus (SSI_NUM_SLAVES)

¹ present when SSI_IS_MASTER = 1

² present when SSI_IS_MASTER = 0

³ present when SSI_INTR_IO = Combined Interrupt (1)

⁴ present when SSI_INTR_IO = Individual Interrupts (0) AND SSI_INTR_POL = Active High (1)

⁵ present when SSI_INTR_IO = Individual Interrupts (0) AND SSI_INTR_POL = Active Low (0)

⁶ present when SSI_CLK_EN_MODE = 1

⁷ present when SSI_HAS_DMA = 1

Signals in red indicate a registered input or output.

5.2 DW_apb_ssi Signal Descriptions

Table 5-1 identifies the signals that are associated with the DW_apb_ssi.



Note

The Description column in Table 5-1 provides detailed information about each signal.

- In the **Registered** field, a “Yes” indicates whether an I/O signal is directly connected to an internal register and nothing else. An I/O signal is also considered to be registered if the signal is connected to one or more inverters or buffers between the I/O port and internal register, but not connected to any logic that involves another signal.
- The **Input/Output Delay** field provides the percentage of the clock cycle assumed to be used by logic outside this design. The given value is used to automatically define the default synthesis constraints for input/output delay. You can override these default values in the Specify Port Constraints activity in coreConsultant or coreAssembler.

Table 5-1 DW_apb_ssi Signal Description

Name	Width	I/O	Description
APB Slave Interface			
pclk	1 bit	In	APB clock. Registered: N/A Synchronous to: N/A External Input Delay: N/A
preseln	1 bit	In	APB Reset Signal Active State: Low Registered: N/A Synchronous to: Asynchronous APB interface domain reset. This signal resets only the bus interface. The signal is asserted asynchronously, but is deasserted synchronously after the rising edge of pclk. The synchronization must be provided external to this component. External Input Delay: N/A
psel	1 bit	In	APB peripheral select that lasts for two pclk cycles. When asserted, indicates that the peripheral has been selected for read or write operation. Active State: High Registered: No Synchronous to: pclk External Input Delay: 40%
penable	1 bit	In	APB enable control. Asserted for a single pclk cycle and used for timing read or write operations. Active State: High Registered: No Synchronous to: pclk External Input Delay: 70%

Table 5-1 DW_apb_ssi Signal Description (Continued)

Name	Width	I/O	Description
pwrite	1 bit	In	<p>APB write control.</p> <p>Active State: When high, indicates a write access to the peripheral; when low, indicates a read access.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Input Delay: 70%</p>
paddr	8 bits	In	<p>APB address bus. Uses lower 8 bits of the address bus for register decode.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Input Delay: 33%</p>
pwdata	w-1:0	In	<p>APB write data bus. Driven by the bus master (bridge unit) during write cycles. Can be 8, 16, or 32 bits wide.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Input Delay: 70%</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.</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 95%</p>
DMA Interface (only present when configured with DMA interface) refer to DMA Controller Interface			
dma_tx_req	1 bit	Out	<p>Transmit FIFO DMA Request – Asserted when the transmit FIFO requires service from the DMA Controller; that is, the transmit FIFO is at or below the watermark level.</p> <p>0 – not requesting 1 – requesting</p> <p>Software must set up the DMA controller with the number of words to be transferred when a request is made. When using the DW_ahb_dmac, this value is programmed in the SRC_MSIZE field of the CTLx register.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 90%</p>

Table 5-1 DW_apb_ssi Signal Description (Continued)

Name	Width	I/O	Description
dma_rx_req	1 bit	Out	<p>Receive FIFO DMA Request – Asserted when the receive FIFO requires service from the DMA Controller; that is, the receive FIFO is at or above the watermark level.</p> <p>0 – not requesting 1 – requesting</p> <p>Software must set up the DMA controller with the number of words to be transferred when a request is made. When using the DW_ahb_dmac, this value is programmed in the DEST_MSIZ field of the CTLx register.</p> <p>Active State: High Registered: Yes Synchronous to: pclk External Output Delay: 90%</p>
dma_tx_single	1 bit	Out	<p>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 External Output Delay: 90%</p>
dma_rx_single	1 bit	Out	<p>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 External Output Delay: 90%</p>
dma_tx_ack	1 bit	In	<p>DMA Transmit Acknowledgement – Sent by the DMA Controller to acknowledge the end of each DMA burst or single transaction to the transmit FIFO.</p> <p>Active State: High Registered: No Synchronous to: pclk External Input Delay: 5%</p>
dma_rx_ack	1 bit	In	<p>DMA Receive Acknowledgement – Sent by the DMA controller to acknowledge the end of each DMA burst or single transaction from the receive FIFO.</p> <p>Active State: High Registered: No Synchronous to: pclk External Input Delay: 5%</p>

Table 5-1 DW_apb_ssi Signal Description (Continued)

Name	Width	I/O	Description
Serial Interface			
ssi_clk	1 bit	In	Peripheral serial clock signal. Registered: N/A Synchronous to: N/A External Input Delay: N/A
ssi_rst_n	1 bit	In	Peripheral reset signal. Active State: Low Registered: N/A Synchronous to: N/A External Input Delay: N/A
txd	1 bit	Out	Transmit data signal. Output data from the serial master or serial slave is transmitted on this line. Active State: High Registered: Yes Synchronous to: ssi_clk External Output Delay: 92%
rxn	1 bit	In	Receive data signal. Input data from a serial-master or serial-slave device is received on this line. Active State: High Registered: No Synchronous to: ssi_clk External Input Delay: 87%
ss_in_n	1 bit	In	Slave select input. When configured as a serial slave, this signal selects the device. When configured as a serial master, this signal can be used to inform the system of master contention on the bus. Active State: Low Registered: No Synchronous to: ssi_clk External Input Delay: 92%
ssi_oe_n	1 bit	Out	Output enable signal. Used to control external buffers on serial output lines. Active State: Low Registered: Yes Synchronous to: ssi_clk External Output Delay: 92%

Table 5-1 DW_apb_ssi Signal Description (Continued)

Name	Width	I/O	Description
ssi_clk_en	1 bit	In	<p><i>Optional.</i> Enable signal for ssi_clk. When the ssi_clk input is connected to pclk, all flip-flops clocked by the ssi_clk propagate only when this signal is active, which gives you control of the frequency ratio between pclk and ssi_clk. This signal is used only when SSI_CLK_EN_MODE = 1.</p> <p>Active State: High</p> <p>Registered: No</p> <p>Synchronous to: N/A</p> <p>External Input Delay: N/A</p>
ssi_sleep	1 bit	Out	<p>SSI enable flag. This signal is active when the DW_apb_ssi is enabled. It can be used by the system clock generator/controller module in order to disable the ssi_clk input of the DW_apb_ssi; this reduces power consumption in the system.</p> <p>0: Not safe to remove ssi_clk 1: Safe to remove ssi_clk</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 90%</p>
Master Interface (Present only when configured as a serial master)			
sclk_out	1 bit	Out	<p>Serial bit-rate clock. Generated by the DW_apb_ssi from ssi_clk.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: ssi_clk</p> <p>External Output Delay: 92%</p>
ss_0_n to ss_15_n	1 bit per signal	Out	<p>Slave select output. For MWire and SPI, the active polarity is logic 0; for SSP, the active polarity is logic 1.</p> <p>The number of slave select signals depends on the number of configured slaves (SSI_NUM_SLAVES). For example, if you have four slaves configured (SSI_NUM_SLAVES = 4), the following signals are generated:</p> <p>ss_0_n ss_1_n ss_2_n ss_3_n</p> <p>Active State: Low</p> <p>Registered: Yes</p> <p>Synchronous to: ssi_clk</p> <p>External Output Delay: 92%</p>

Table 5-1 DW_apb_ssi Signal Description (Continued)

Name	Width	I/O	Description
Slave Interface (Present only when configured as a serial slave)			
sclk_in	1 bit	In	<p>Serial bit-rate clock. Generated by the serial bus master and used by the DW_apb_ssi slave to regulate data transfer. Never used to clock any registers; instead, an edge-detector is used in order for everything to run in sclk domain.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: This signal is asynchronous to the ssi_clk.</p> <p>External Input Delay: N/A</p>
Interrupt Signals			
ssi_intr/ ssi_intr_n	1 bit	Out	<p><i>Optional.</i> Combined SSI interrupt flag. Logical OR of all individual interrupts.</p> <p>Active State: Configurable active polarity.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: N/A</p> <p>Dependencies: Present only when SSI_INTR_IO = Combined (1).</p>
ssi_txe_intr/ ssi_txe_intr_n	1 bit	Out	<p><i>Optional.</i> Transmit FIFO empty interrupt. Active when the transmit FIFO is equal to or below the threshold value (TFT).</p> <p>Active State: Configurable active polarity.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 80%</p> <p>Dependencies: Present only when SSI_INTR_IO = Individual (0).</p>
ssi_txo_intr/ ssi_txo_intr_n	1 bit	Out	<p><i>Optional.</i> Transmit FIFO overflow interrupt. Active when the APB attempts to write to a full transmit FIFO.</p> <p>Active State: Configurable active polarity.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 80%</p> <p>Dependencies: Present only when SSI_INTR_IO = Individual (0)</p>
ssi_rxf_intr/ ssi_rxf_intr_n	1 bit	Out	<p><i>Optional.</i> Receive FIFO full interrupt. Active when the receive FIFO is equal to or above the threshold value (RFT) plus 1.</p> <p>Active State: Configurable active polarity.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 80%</p> <p>Dependencies: Present only when SSI_INTR_IO = Individual (0).</p>

Table 5-1 DW_apb_ssi Signal Description (Continued)

Name	Width	I/O	Description
ssi_rxo_intr/ ssi_rxo_intr_n	1 bit	Out	<p><i>Optional.</i> Receive FIFO overflow interrupt. Active when the receive logic attempts to write into a full receive FIFO.</p> <p>Active State: Configurable active polarity.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 80%</p> <p>Dependencies: Present only when SSI_INTR_IO = Individual (0).</p>
ssi_rxu_intr/ ssi_rxu_intr_n	1 bit	Out	<p><i>Optional.</i> Receive FIFO underflow interrupt. Active when the APB attempts to read from an empty receive FIFO.</p> <p>Active State: Configurable active polarity.</p> <p>Registered: No</p> <p>Synchronous to: 80%</p> <p>External Output Delay: pclk</p> <p>Dependencies: Present only when SSI_INTR_IO = Individual (0).</p>
ssi_mst_intr/ ssi_mst_intr_n	1bit	Out	<p><i>Optional.</i> Multi-master contention interrupt. Informs of possible bus contention.</p> <p>Active State: Configurable active polarity.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 80%</p> <p>Dependencies: Present only when SSI_INTR_IO = Individual (0).</p>

6

Registers

This section describes the programmable registers of the DW_apb_ssi.

6.1 Register Memory Map

Table 6-1 provides the details of the DW_apb_ssi memory map. All registers in the DW_apb_ssi are addressed at 32-bit boundaries to remain consistent with the AHB bus. Where the physical size of any register is less than 32-bits wide, the upper unused bits of the 32-bit boundary are reserved. Writing to these bits has no effect; reading from these bits returns 0.

Table 6-1 Memory Map of DW_apb_ssi

Name	Address Offset	Width	Description
CTRLR0	0x0	16 bits	Control Register 0 Reset Value: Configuration Dependent for some bit fields
CTRLR1	0x04	16 bits	Control Register 1 Reset Value: 0x0
SSIENR	0x08	1 bit	SSI Enable Register Reset Value: 0x0
MWCR	0x0C	3 bits	Microwire Control Register Reset Value: 0x0
SER	0x10	See Description	Slave Enable Register Width: <i>SSI_NUM_SLAVES</i> Reset Value: 0x0
BAUDR	0x14	16 bits	Baud Rate Select Reset Value: 0x0
TXFTLR	0x18	<i>TX_ABW</i>	Transmit FIFO Threshold Level Reset Value: 0x0
RXFTLR	0x1C	<i>RX_ABW</i>	Receive FIFO Threshold Level Reset Value: 0x0

Table 6-1 Memory Map of DW_apb_ssi (Continued)

Name	Address Offset	Width	Description
TXFLR	0x20	See Description	Transmit FIFO Level Register Width: TX_ABW+1 Reset Value: 0x0
RXFLR	0x24	See Description	Receive FIFO Level Register Width: RX_ABW+1 Reset Value: 0x0
SR	0x28	7 bits	Status Register Reset Value: 0x6
IMR	0x2C	See Description	Interrupt Mask Register Width: 6 bits: when SSI_IS_MASTER = 1) 5 bits: when SSI_IS_MASTER = 0) Reset Value: 0x3F/0x1F
ISR	0x30	6 bits	Interrupt Status Register Reset Value: 0x0
RISR	0x34	6 bits	Raw Interrupt Status Register Reset Value: 0x0
TXOICR	0x38	1 bit	Transmit FIFO Overflow Interrupt Clear Register Reset Value: 0x0
RXOICR	0x3C	1 bit	Receive FIFO Overflow Interrupt Clear Register Reset Value: 0x0
RXUICR	0x40	1 bit	Receive FIFO Underflow Interrupt Clear Register Reset Value: 0x0
MSTICR	0x44	1 bit	Multi-Master Interrupt Clear Register Reset Value: 0x0
ICR	0x48	1 bit	Interrupt Clear Register Reset Value: 0x0
DMACR	0x4C	2 bits	DMA Control Register Reset Value: 0x0
DMATDLR	0x50	TX_ABW	DMA Transmit Data Level Reset Value: 0x0
DMARDLR	0x54	RX_ABW	DMA Receive Data Level Reset Value: 0x0
IDR	0x58	32 bits	Identification Register Reset Value: Not affected by reset

Table 6-1 Memory Map of DW_apb_ssi (Continued)

Name	Address Offset	Width	Description
SSI_COMP_VERSION	0x5C	32 bits	coreKit version ID register Reset Value: See the releases table in the AMBA 2 release notes
DR	0x60 - 0xec	16 bits	Data Register Reset Value: 0x0 NOTE: If the Data Register (DR) is accessed from an AHB master (such as a DMA controller or a processor), the AHB transfer type may be a burst. During AHB burst transfers, the address increments after each beat of the burst. To facilitate an AHB burst, read, or write operation to the transmit or receive FIFO, the Data Register occupies thirty-six 32-bit address locations of memory map. Each of the 16-bit address locations are aliased to the DR; single accesses to the DR may use any of the 16-bit address locations.
RX_SAMPLE_DLY	0xf0	8 bits	RXD Sample Delay Register Reset Value: 0x0
RSVD_0	0xf4	32 bits	Reserved location for future use Reset Value: Writes have no effect; reads return value of 0
RSVD_1	0xf8	32 bits	Reserved location for future use Reset Value: Writes have no effect; reads return value of 0
RSVD_2	0xfc	32 bits	Reserved location for future use Reset Value: Writes have no effect; reads return value of 0

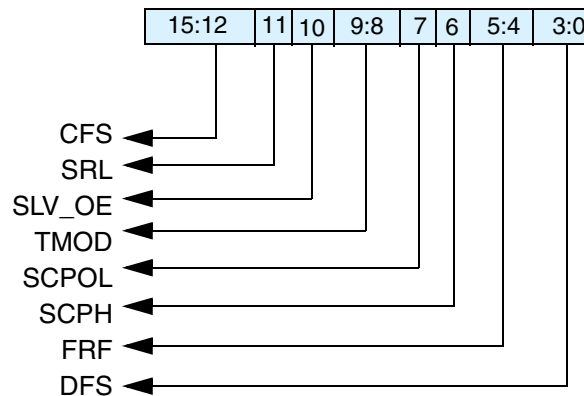
6.2 Register and Field Descriptions

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

6.2.1 CTRLR0

- ❖ **Name:** Control Register 0
- ❖ **Size:** 16 bits
- ❖ **Address Offset:** 0x0
- ❖ **Read/write access:** read/write

This register controls the serial data transfer. It is impossible to write to this register when the DW_apb_ssi is enabled. The DW_apb_ssi is enabled and disabled by writing to the [SSIENR](#) register.



Bits	Name	R/W	Description
15:12	CFS	R/W	Control Frame Size. Selects the length of the control word for the Microwire frame format. For the field decode, refer to Table 6-3 on page 97. Reset Value: 0x0
11	SRL	R/W	Shift Register Loop. Used for testing purposes only. When internally active, connects the transmit shift register output to the receive shift register input. Can be used in both serial-slave and serial-master modes. 0 – Normal Mode Operation 1 – Test Mode Operation When the DW_apb_ssi is configured as a slave in loopback mode, the ss_in_n and ssi_clk signals must be provided by an external source. In this mode, the slave cannot generate these signals because there is nothing to which to loop back. Reset Value: 0x0

Bits	Name	R/W	Description
10	SLV_OE	R/W	<p>Slave Output Enable. Relevant only when the DW_apb_ssi is configured as a serial-slave device. When configured as a serial master, this bit field has no functionality. This bit enables or disables the setting of the ssi_oe_n output from the DW_apb_ssi serial slave. When SLV_OE = 1, the ssi_oe_n output can never be active. When the ssi_oe_n output controls the tri-state buffer on the txd output from the slave, a high impedance state is always present on the slave txd output when SLV_OE = 1.</p> <p>This is useful when the master transmits in broadcast mode (master transmits data to all slave devices). Only one slave may respond with data on the master rxd line. This bit is enabled after reset and must be disabled by software (when broadcast mode is used), if you do not want this device to respond with data.</p> <p>0 — Slave txd is enabled 1 — Slave txd is disabled</p> <p>Reset Value: 0x0</p>
9:8	TMOD	R/W	<p>Transfer Mode. Selects the mode of transfer for serial communication. This field does not affect the transfer duplicity. Only indicates whether the receive or transmit data are valid. In transmit-only mode, data received from the external device is not valid and is not stored in the receive FIFO memory; it is overwritten on the next transfer.</p> <p>In receive-only mode, transmitted data are not valid. After the first write to the transmit FIFO, the same word is retransmitted for the duration of the transfer.</p> <p>In transmit-and-receive mode, both transmit and receive data are valid. The transfer continues until the transmit FIFO is empty. Data received from the external device are stored into the receive FIFO memory, where it can be accessed by the host processor.</p> <p>In eeprom-read mode, receive data is not valid while control data is being transmitted. When all control data is sent to the EEPROM, receive data becomes valid and transmit data becomes invalid. All data in the transmit FIFO is considered control data in this mode. This transfer mode is only valid when the DW_apb_ssi is configured as a master device.</p> <p>00 — Transmit & Receive 01 — Transmit Only 10 — Receive Only 11 — EEPROM Read</p> <p>Reset Value: 0x0</p>
7	SCPOL	R/W	<p>Serial Clock Polarity. Valid when the frame format (FRF) is set to Motorola SPI. Used to select the polarity of the inactive serial clock, which is held inactive when the DW_apb_ssi master is not actively transferring data on the serial bus.</p> <p>0 – Inactive state of serial clock is low 1 – Inactive state of serial clock is high</p> <p>Dependencies: When SSI_HC_FRF=1, SCPOL bit is a read-only bit with its value set by SSI_DFLT_SCPOL.</p> <p>Reset Value: SSI_DFLT_SCPOL</p>

Bits	Name	R/W	Description
6	SCPH	R/W	<p>Serial Clock Phase. Valid when the frame format (FRF) is set to Motorola SPI. The serial clock phase selects the relationship of the serial clock with the slave select signal.</p> <p>When SCPH = 0, data are captured on the first edge of the serial clock. When SCPH = 1, the serial clock starts toggling one cycle after the slave select line is activated, and data are captured on the second edge of the serial clock.</p> <p>0: Serial clock toggles in middle of first data bit 1: Serial clock toggles at start of first data bit</p> <p>Dependencies: When SSI_HC_FRF=1, SCPH bit is a read-only bit, with its value set by SSI_DFLT_SCPH.</p> <p>Reset Value: SSI_DFLT_SCPH</p>
5:4	FRF	R/W	<p>Frame Format. Selects which serial protocol transfers the data.</p> <p>00 — Motorola SPI 01 — Texas Instruments SSP 10 — National Semiconductors Microwire 11 — Reserved</p> <p>Dependencies: When SSI_HC_FRF=1, FRF is read-only and its value is set by SSI_DFLT_FRF.</p> <p>Reset Value: SSI_DFLT_FRF</p>
3:0	DFS	R/W	<p>Data Frame Size. Selects the data frame length. When the data frame size is programmed to be less than 16 bits, the receive data are automatically right-justified by the receive logic, with the upper bits of the receive FIFO zero-padded.</p> <p>You must right-justify transmit data before writing into the transmit FIFO. The transmit logic ignores the upper unused bits when transmitting the data. For the field decode, refer to Table 6-2.</p> <p>Reset Value: 0x7</p>

Table 6-2 DFS Decode

DFS Value	Description
0000	Reserved – undefined operation
0001	Reserved – undefined operation
0010	Reserved – undefined operation
0011	4-bit serial data transfer
0100	5-bit serial data transfer
0101	6-bit serial data transfer
0110	7-bit serial data transfer
0111	8-bit serial data transfer
1000	9-bit serial data transfer
1001	10-bit serial data transfer

Table 6-2 DFS Decode (Continued)

DFS Value	Description
1010	11-bit serial data transfer
1011	12-bit serial data transfer
1100	13-bit serial data transfer
1101	14-bit serial data transfer
1110	15-bit serial data transfer
1111	16-bit serial data transfer

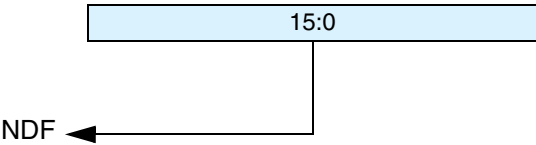
Table 6-3 CFS Decode

CFS Value	Description
0000	1-bit control word
0001	2-bit control word
0010	3-bit control word
0011	4-bit control word
0100	5-bit control word
0101	6-bit control word
0110	7-bit control word
0111	8-bit control word
1000	9-bit control word
1001	10-bit control word
1010	11-bit control word
1011	12-bit control word
1100	13-bit control word
1101	14-bit control word
1110	15-bit control word
1111	16-bit control word

6.2.2 CTRLR1

- ❖ **Name:** Control Register 1
- ❖ **Size:** 16 bits
- ❖ **Address Offset:** 0x04
- ❖ **Read/write access:** read/write

This register exists only when the DW_apb_ssi is configured as a master device. When the DW_apb_ssi is configured as a serial slave, writing to this location has no effect; reading from this location returns 0. Control register 1 controls the end of serial transfers when in receive-only mode. It is impossible to write to this register when the DW_apb_ssi is enabled. The DW_apb_ssi is enabled and disabled by writing to the [SSIENR](#) register.

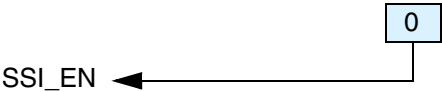


Bits	Name	R/W	Description
15:0	NDF	R/W	<p>Number of Data Frames. When TMOD = 10 or TMOD = 11, this register field sets the number of data frames to be continuously received by the DW_apb_ssi. The DW_apb_ssi continues to receive serial data until the number of data frames received is equal to this register value plus 1, which enables you to receive up to 64 KB of data in a continuous transfer.</p> <p>When the DW_apb_ssi is configured as a serial slave, the transfer continues for as long as the slave is selected. Therefore, this register serves no purpose and is not present when the DW_apb_ssi is configured as a serial slave.</p> <p>Reset Value: 0x0</p>

6.2.3 SSIENR

- ❖ **Name:** SSI Enable Register
- ❖ **Size:** 1 bit
- ❖ **Address Offset:** 0x08
- ❖ **Read/write access:** read/write

This register enables and disables the DW_apb_ssi.

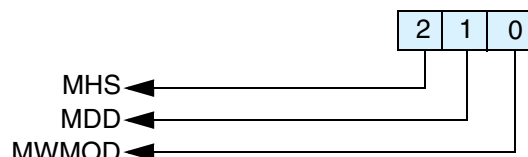


Bits	Name	R/W	Description
0	SSI_EN	R/W	SSI Enable. Enables and disables all DW_apb_ssi operations. When disabled, all serial transfers are halted immediately. Transmit and receive FIFO buffers are cleared when the device is disabled. It is impossible to program some of the DW_apb_ssi control registers when enabled. When disabled, the ssi_sleep output is set (after delay) to inform the system that it is safe to remove the ssi_clk, thus saving power consumption in the system. Reset Value: 0x0

6.2.4 MWCR

- ❖ **Name:** Microwire Control Register
- ❖ **Size:** 3 bits
- ❖ **Address Offset:** 0x0C
- ❖ **Read/write access:** read/write

This register controls the direction of the data word for the half-duplex Microwire serial protocol. It is impossible to write to this register when the DW_apb_ssi is enabled. The DW_apb_ssi is enabled and disabled by writing to the [SSIENR](#) register.

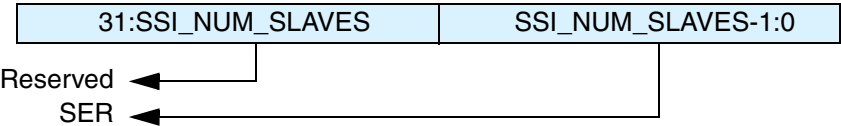


Bits	Name	R/W	Description
2	MHS	R/W	<p>Microwire Handshaking. Relevant only when the DW_apb_ssi is configured as a serial-master device. When configured as a serial slave, this bit field has no functionality. Used to enable and disable the “busy/ready” handshaking interface for the Microwire protocol. When enabled, the DW_apb_ssi checks for a ready status from the target slave, after the transfer of the last data/control bit, before clearing the BUSY status in the SR register.</p> <p>0: handshaking interface is disabled 1: handshaking interface is enabled</p> <p>Reset Value: 0x0</p>
1	MDD	R/W	<p>Microwire Control. Defines the direction of the data word when the Microwire serial protocol is used. When this bit is set to 0, the data word is received by the DW_apb_ssi MacroCell from the external serial device. When this bit is set to 1, the data word is transmitted from the DW_apb_ssi MacroCell to the external serial device.</p> <p>Reset Value: 0x0</p>
0	MWMOD	R/W	<p>Microwire Transfer Mode. Defines whether the Microwire transfer is sequential or non-sequential. When sequential mode is used, only one control word is needed to transmit or receive a block of data words. When non-sequential mode is used, there must be a control word for each data word that is transmitted or received.</p> <p>0 – non-sequential transfer 1 – sequential transfer</p> <p>Reset Value: 0x0</p>

6.2.5 SER

- ❖ **Name:** Slave Enable Register
- ❖ **Size:** SSI_NUM_SLAVES
- ❖ **Address Offset:** 0x10
- ❖ **Read/write access:** read/write

This register is valid only when the DW_apb_ssi is configured as a master device. When the DW_apb_ssi is configured as a serial slave, writing to this location has no effect; reading from this location returns 0. The register enables the individual slave select output lines from the DW_apb_ssi master. Up to 16 slave-select output signals are available on the DW_apb_ssi master. You cannot write to this register when DW_apb_ssi is busy.

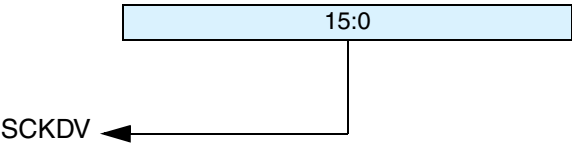


Bits	Name	R/W	Description
31:SSI_NUM_SLAVES	Reserved	N/A	Reserved
SSI_NUM_SLAVES-1:0	SER	R/W	<p>Slave Select Enable Flag. Each bit in this register corresponds to a slave select line (ss_x_n]) from the DW_apb_ssi master. When a bit in this register is set (1), the corresponding slave select line from the master is activated when a serial transfer begins. It should be noted that setting or clearing bits in this register have no effect on the corresponding slave select outputs until a transfer is started. Before beginning a transfer, you should enable the bit in this register that corresponds to the slave device with which the master wants to communicate.</p> <p>When not operating in broadcast mode, only one bit in this field should be set.</p> <p>1: Selected 0: Not Selected</p> <p>Reset Value: 0x0</p>

6.2.6 BAUDR

- ❖ **Name:** Baud Rate Select
- ❖ **Size:** 16 bits
- ❖ **Address Offset:** 0x14
- ❖ **Read/write access:** read/write

This register is valid only when the DW_apb_ssi is configured as a master device. When the DW_apb_ssi is configured as a serial slave, writing to this location has no effect; reading from this location returns 0. The register derives the frequency of the serial clock that regulates the data transfer. The 16-bit field in this register defines the ssi_clk divider value. It is impossible to write to this register when the DW_apb_ssi is enabled. The DW_apb_ssi is enabled and disabled by writing to the [SSIENR](#) register.

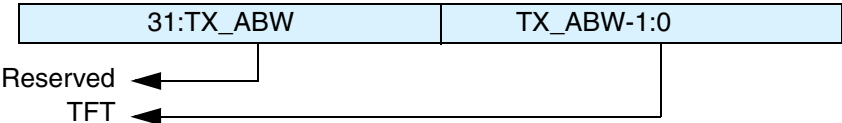


Bits	Name	R/W	Description
15:0	SCKDV	R/W	<p>SSI Clock Divider. The LSB for this field is always set to 0 and is unaffected by a write operation, which ensures an even value is held in this register. If the value is 0, the serial output clock (sclk_out) is disabled. The frequency of the sclk_out is derived from the following equation:</p> $\frac{F_{\text{sclk_out}}}{\text{SCKDV}} = F_{\text{ssi_clk}}$ <p>where SCKDV is any even value between 2 and 65534. For example:</p> <p>for Fssi_clk = 3.6864MHz and SCKDV =2</p> $F_{\text{sclk_out}} = 3.6864/2 = 1.8432\text{MHz}$ <p>Reset Value: 0x0</p>

6.2.7 TXFTLR

- ❖ **Name:** Transmit FIFO Threshold Level
- ❖ **Size:** TX_ABW
- ❖ **Address Offset:** 0x18
- ❖ **Read/write access:** read/write

This register controls the threshold value for the transmit FIFO memory. It is impossible to write to this register when the DW_apb_ssi is enabled. The DW_apb_ssi is enabled and disabled by writing to the [SSIENR](#) register.



Bits	Name	R/W	Description
31:TX_ABW	Reserved	N/A	Reserved
TX_ABW-1:0	TFT	R/W	<p>Transmit FIFO Threshold. Controls the level of entries (or below) at which the transmit FIFO controller triggers an interrupt. The FIFO depth is configurable in the range 2-256; this register is sized to the number of address bits needed to access the FIFO.</p> <p>If you attempt to set this value greater than or equal to the depth of the FIFO, this field is not written and retains its current value. When the number of transmit FIFO entries is less than or equal to this value, the transmit FIFO empty interrupt is triggered. For field decode, refer to Table 6-4.</p> <p>Reset Value: 0x0</p>

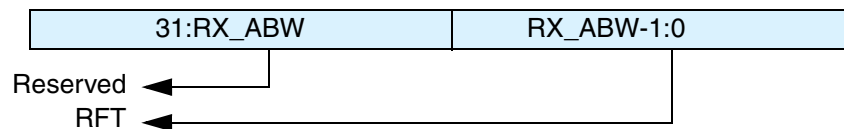
Table 6-4 TFT Decode

TFT Value	Description
0000_0000	ssi_txe_intr is asserted when 0 data entries are present in transmit FIFO
0000_0001	ssi_txe_intr is asserted when 1 or less data entry is present in transmit FIFO
0000_0010	ssi_txe_intr is asserted when 2 or less data entries are present in transmit FIFO
0000_0011	ssi_txe_intr is asserted when 3 or less data entries are present in transmit FIFO
:	:
:	:
1111_1100	ssi_txe_intr is asserted when 252 or less data entries are present in transmit FIFO
1111_1101	ssi_txe_intr is asserted when 253 or less data entries are present in transmit FIFO
1111_1110	ssi_txe_intr is asserted when 254 or less data entries are present in transmit FIFO
1111_1111	ssi_txe_intr is asserted when 255 or less data entries are present in transmit FIFO

6.2.8 RXFTLR

- ❖ **Name:** Receive FIFO Threshold Level
- ❖ **Size:** *RX_ABW*
- ❖ **Address Offset:** 0x1C
- ❖ **Read/write access:** read/write

This register controls the threshold value for the receive FIFO memory. It is impossible to write to this register when the DW_apb_ssi is enabled. The DW_apb_ssi is enabled and disabled by writing to the [SSIENR](#) register.



Bits	Name	R/W	Reset	Description
31:RX_ABW	Reserved	N/A	N/A	Reserved
RX_ABW-1:0	RFT	R/W		<p>Receive FIFO Threshold. Controls the level of entries (or above) at which the receive FIFO controller triggers an interrupt. The FIFO depth is configurable in the range 2-256. This register is sized to the number of address bits needed to access the FIFO. If you attempt to set this value greater than the depth of the FIFO, this field is not written and retains its current value.</p> <p>When the number of receive FIFO entries is greater than or equal to this value + 1, the receive FIFO full interrupt is triggered. For field decode, refer to Table 6-5.</p> <p>Reset Value: 0x0</p>

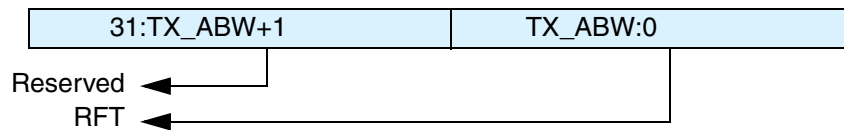
Table 6-5 RFT

RFT Value	Description
0000_0000	ssi_rxf_intr is asserted when 1 or more data entry is present in receive FIFO
0000_0001	ssi_rxf_intr is asserted when 2 or more data entries are present in receive FIFO
0000_0010	ssi_rxf_intr is asserted when 3 or more data entries are present in receive FIFO
0000_0011	ssi_rxf_intr is asserted when 4 or more data entries are present in receive FIFO
:	:
:	:
1111_1100	ssi_rxf_intr is asserted when 253 or more data entries are present in receive FIFO
1111_1101	ssi_rxf_intr is asserted when 254 or more data entries are present in receive FIFO
1111_1110	ssi_rxf_intr is asserted when 255 or more data entries are present in receive FIFO
1111_1111	ssi_rxf_intr is asserted when 256 data entries are present in receive FIFO

6.2.9 TXFLR

- ❖ **Name:** Transmit FIFO Level Register
- ❖ **Size:** $TX_ABW + 1$
- ❖ **Address Offset:** 0x20
- ❖ **Read/write access:** read-only

This register contains the number of valid data entries in the transmit FIFO memory.

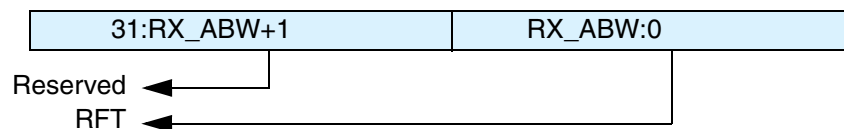


Bits	Name	R/W	Description
31:TX_ABW+1	Reserved	N/A	Reserved
TX_ABW:0	TXTFL	R	Transmit FIFO Level. Contains the number of valid data entries in the transmit FIFO. Reset Value: 0x0

6.2.10 RXFLR

- ❖ **Name:** Receive FIFO Level Register
- ❖ **Size:** $RX_ABW + 1$
- ❖ **Address Offset:** 0x24
- ❖ **Read/write access:** read-only

This register contains the number of valid data entries in the receive FIFO memory. This register can be read at any time.

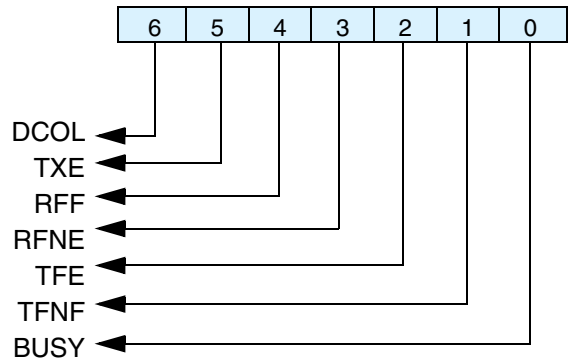


Bits	Name	R/W	Description
31:RX_ABW+1	Reserved	N/A	Reserved
RX_ABW:0	RXTFL	R	Receive FIFO Level. Contains the number of valid data entries in the receive FIFO. Reset Value: 0x0

6.2.11 SR

- ❖ **Name:** Status Register
- ❖ **Size:** 7 bits
- ❖ **Address Offset:** 0x28
- ❖ **Read/write access:** read-only

This is a read-only register used to indicate the current transfer status, FIFO status, and any transmission/reception errors that may have occurred. The status register may be read at any time. None of the bits in this register request an interrupt.



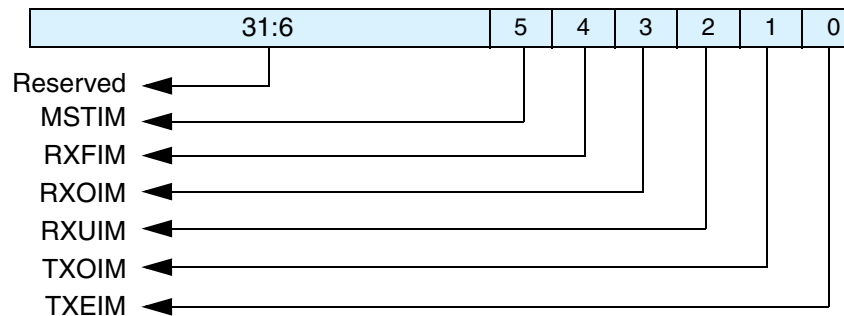
Bits	Name	R/W	Description
6	DCOL	R	<p>Data Collision Error. Relevant only when the DW_apb_ssi is configured as a master device. This bit is set if the DW_apb_ssi master is actively transmitting when another master selects this device as a slave. This informs the processor that the last data transfer was halted before completion. This bit is cleared when read.</p> <p>0 – No error 1 – Transmit data collision error</p> <p>Reset Value: 0x0</p>
5	TXE	R	<p>Transmission Error. Set if the transmit FIFO is empty when a transfer is started. This bit can be set only when the DW_apb_ssi is configured as a slave device. Data from the previous transmission is resent on the txd line. This bit is cleared when read.</p> <p>0 – No error 1 – Transmission error</p> <p>Reset Value: 0x0</p>
4	RFF	R	<p>Receive FIFO 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> <p>0 – Receive FIFO is not full 1 – Receive FIFO is full</p> <p>Reset Value: 0x0</p>

Bits	Name	R/W	Description
3	RFNE	R	<p>Receive FIFO Not Empty. Set when the receive FIFO contains one or more entries and is cleared when the receive FIFO is empty. This bit can be polled by software to completely empty the receive FIFO.</p> <p>0 – Receive FIFO is empty 1 – Receive FIFO is not empty</p> <p>Reset Value: 0x0</p>
2	TFE	R	<p>Transmit FIFO Empty. When the transmit FIFO is completely empty, this bit is set. When the transmit FIFO contains one or more valid entries, this bit is cleared. This bit field does not request an interrupt.</p> <p>0 – Transmit FIFO is not empty 1 – Transmit FIFO is empty</p> <p>Reset Value: 0x1</p>
1	TFNF	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> <p>0 – Transmit FIFO is full 1 – Transmit FIFO is not full</p> <p>Reset Value: 0x1</p>
0	BUSY	R	<p>SSI Busy Flag. When set, indicates that a serial transfer is in progress; when cleared indicates that the DW_apb_ssi is idle or disabled.</p> <p>0 – DW_apb_ssi is idle or disabled 1 – DW_apb_ssi is actively transferring data</p> <p>Reset Value: 0x0</p>

6.2.12 IMR

- ❖ **Name:** Interrupt Mask Register
- ❖ **Size:**
 - 6 bits: when SSI_IS_MASTER = 1)
 - 5 bits: when SSI_IS_MASTER = 0)
- ❖ **Address Offset:** 0x2C
- ❖ **Read/write access:** read/write

This read/write register masks or enables all interrupts generated by the DW_apb_ssi. When the DW_apb_ssi is configured as a slave device, the MSTIM bit field is not present. This changes the reset value from 0x3F for serial-master configurations to 0x1F for serial-slave configurations.



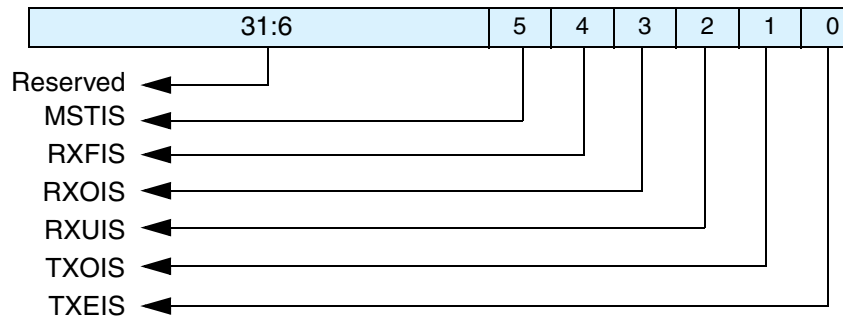
Bits	Name	R/W	Description
31:6	Reserved	N/A	Reserved
5	MSTIM	R/W	Multi-Master Contention Interrupt Mask. This bit field is not present if the DW_apb_ssi is configured as a serial-slave device. 0 – ssi_mst_intr interrupt is masked 1 – ssi_mst_intr interrupt is not masked Reset Value: 0x1
4	RXFIM	R/W	Receive FIFO Full Interrupt Mask 0 – ssi_rxf_intr interrupt is masked 1 – ssi_rxf_intr interrupt is not masked Reset Value: 0x1
3	RXOIM	R/W	Receive FIFO Overflow Interrupt Mask 0 – ssi_rxo_intr interrupt is masked 1 – ssi_rxo_intr interrupt is not masked Reset Value: 0x1
2	R XUIM	R/W	Receive FIFO Underflow Interrupt Mask 0 – ssi_rxu_intr interrupt is masked 1 – ssi_rxu_intr interrupt is not masked Reset Value: 0x1

Bits	Name	R/W	Description
1	TXOIM	RW	Transmit FIFO Overflow Interrupt Mask 0 – ssi_txo_intr interrupt is masked 1 – ssi_txo_intr interrupt is not masked Reset Value: 0x1
0	TXEIM	RW	Transmit FIFO Empty Interrupt Mask 0 – ssi_txe_intr interrupt is masked 1 – ssi_txe_intr interrupt is not masked Reset Value: 0x1

6.2.13 ISR

- ❖ **Name:** Interrupt Status Register
- ❖ **Size:** 6 bits
- ❖ **Address Offset:** 0x30
- ❖ **Read/write access:** read-only

This register reports the status of the DW_apb_ssi interrupts after they have been masked.



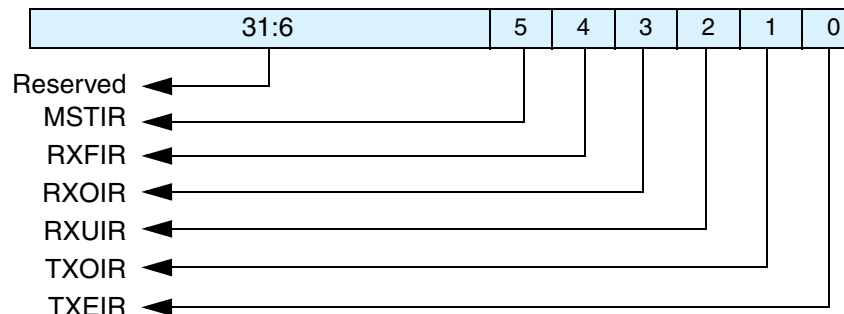
Bits	Name	R/W	Description
31:6	Reserved	N/A	Reserved
5	MSTIS	R	Multi-Master Contention Interrupt Status. This bit field is not present if the DW_apb_ssi is configured as a serial-slave device. 0 = ssi_mst_intr interrupt not active after masking 1 = ssi_mst_intr interrupt is active after masking Reset Value: 0x0
4	RXFIS	R	Receive FIFO Full Interrupt Status 0 = ssi_rxf_intr interrupt is not active after masking 1 = ssi_rxf_intr interrupt is full after masking Reset Value: 0x0
3	RXOIS	R	Receive FIFO Overflow Interrupt Status 0 = ssi_rxo_intr interrupt is not active after masking 1 = ssi_rxo_intr interrupt is active after masking Reset Value: 0x0

Bits	Name	R/W	Description
2	RXUIS	R	Receive FIFO Underflow Interrupt Status 0 = ssi_rxu_intr interrupt is not active after masking 1 = ssi_rxu_intr interrupt is active after masking Reset Value: 0x0
1	TXOIS	R	Transmit FIFO Overflow Interrupt Status 0 = ssi_txo_intr interrupt is not active after masking 1 = ssi_txo_intr interrupt is active after masking Reset Value: 0x0
0	TXEIS	R	Transmit FIFO Empty Interrupt Status 0 = ssi_txe_intr interrupt is not active after masking 1 = ssi_txe_intr interrupt is active after masking Reset Value: 0x0

6.2.14 RISR

- ❖ **Name:** Raw Interrupt Status Register
- ❖ **Size:** 32 bits
- ❖ **Address Offset:** 0x34
- ❖ **Read/write access:** read-only

This read-only register reports the status of the DW_apb_ssi interrupts prior to masking.



Bits	Name	R/W	Description
31:6	Reserved	N/A	Reserved
5	MSTIR	R	Multi-Master Contention Raw Interrupt Status. This bit field is not present if the DW_apb_ssi is configured as a serial-slave device. 0 = ssi_mst_intr interrupt is not active prior to masking 1 = ssi_mst_intr interrupt is active prior masking Reset Value: 0x0
4	RXFIR	R	Receive FIFO Full Raw Interrupt Status 0 = ssi_rxf_intr interrupt is not active prior to masking 1 = ssi_rxf_intr interrupt is active prior to masking Reset Value: 0x0

Bits	Name	R/W	Description
3	RXOIR	R	Receive FIFO Overflow Raw Interrupt Status 0 = ssi_rxo_intr interrupt is not active prior to masking 1 = ssi_rxo_intr interrupt is active prior masking Reset Value: 0x0
2	RXUIR	R	Receive FIFO Underflow Raw Interrupt Status 0 = ssi_rxu_intr interrupt is not active prior to masking 1 = ssi_rxu_intr interrupt is active prior to masking Reset Value: 0x0
1	TXOIR	R	Transmit FIFO Overflow Raw Interrupt Status 0 = ssi_txo_intr interrupt is not active prior to masking 1 = ssi_txo_intr interrupt is active prior masking Reset Value: 0x0
0	TXEIR	R	Transmit FIFO Empty Raw Interrupt Status 0 = ssi_txe_intr interrupt is not active prior to masking 1 = ssi_txe_intr interrupt is active prior masking Reset Value: 0x0

6.2.15 TXOICR

- ❖ **Name:** Transmit FIFO Overflow Interrupt Clear Register
- ❖ **Size:** 1 bit
- ❖ **Address Offset:** 0x38
- ❖ **Read/write access:** read-only



Bits	Name	R/W	Description
0	TXOICR	R	Clear Transmit FIFO Overflow Interrupt. This register reflects the status of the interrupt. A read from this register clears the ssi_txo_intr interrupt; writing has no effect. Reset Value: 0x0

6.2.16 RXOICR

- ❖ **Name:** Receive FIFO Overflow Interrupt Clear Register
- ❖ **Size:** 1 bit
- ❖ **Address Offset:** 0x3C
- ❖ **Read/write access:** read-only



Bits	Name	R/W	Description
0	RXOICR	R	Clear Receive FIFO Overflow Interrupt. This register reflects the status of the interrupt. A read from this register clears the ssi_rxo_intr interrupt; writing has no effect. Reset Value: 0x0

6.2.17 RXUICR

- ❖ **Name:** Receive FIFO Underflow Interrupt Clear Register
- ❖ **Size:** 1 bit
- ❖ **Address Offset:** 0x40
- ❖ **Read/write access:** read-only



Bits	Name	R/W	Description
0	RXUICR	R	Clear Receive FIFO Underflow Interrupt. This register reflects the status of the interrupt. A read from this register clears the ssi_rxu_intr interrupt; writing has no effect. Reset Value: 0x0

6.2.18 MSTICR

- ❖ **Name:** Multi-Master Interrupt Clear Register
- ❖ **Size:** 1 bit
- ❖ **Address Offset:** 0x44
- ❖ **Read/write access:** read-only



Bits	Name	R/W	Description
0	MSTICR	R	Clear Multi-Master Contention Interrupt. This register reflects the status of the interrupt. A read from this register clears the ssi_mst_intr interrupt; writing has no effect. Reset Value: 0x0

6.2.19 ICR

- ❖ **Name:** Interrupt Clear Register
- ❖ **Size:** 1 bit
- ❖ **Address Offset:** 0x48
- ❖ **Read/write access:** read-only



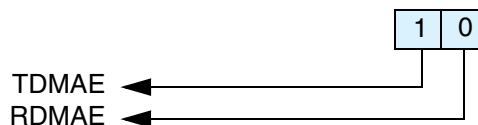
Bits	Name	R/W	Description
0	ICR	R	Clear Interrupts. This register is set if any of the interrupts below are active. A read clears the ssi_txo_intr, ssi_rxu_intr, ssi_rxo_intr, and the ssi_mst_intr interrupts. Writing to this register has no effect. Reset Value: 0x0

6.2.20 DMACR

- ❖ **Name:** DMA Control Register
- ❖ **Size:** 2 bits
- ❖ **Address Offset:** 0x4C
- ❖ **Read/write access:** read/write

This register is only valid when DW_apb_ssi is configured with a set of DMA Controller interface signals (SSI_HAS_DMA = 1). When DW_apb_ssi is not configured for DMA operation, this register will not exist

and writing to the register's address will have no effect; reading from this register address will return zero. The register is used to enable the DMA Controller interface operation.

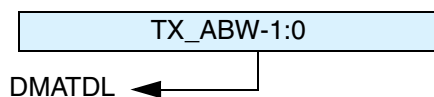


Bits	Name	R/W	Description
1	TDMAE	R/W	Transmit DMA Enable. This bit enables/disables the transmit FIFO DMA channel. 0 = Transmit DMA disabled 1 = Transmit DMA enabled Reset Value: 0x0
0	RDMAE	R/W	Receive DMA Enable. This bit enables/disables the receive FIFO DMA channel 0 = Receive DMA disabled 1 = Receive DMA enabled Reset Value: 0x0

6.2.21 DMATDLR

- ❖ **Name:** DMA Transmit Data Level
- ❖ **Size:** TX_ABW
- ❖ **Address Offset:** 0x50
- ❖ **Read/write access:** read/write

This register is only valid when the DW_apb_ssi is configured with a set of DMA interface signals (SSI_HAS_DMA = 1). When DW_apb_ssi is not configured for DMA operation, this register will not exist and writing to its address will have no effect; reading from its address will return zero.



Bits	Name	R/W	Description
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. Refer to Table 6-6 for the field decode. Reset Value: 0x0

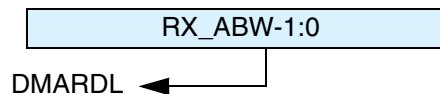
Table 6-6 DMATDL Decode Value

DMATDL Value	Description
0000_0000	dma_tx_req is asserted when 0 data entries are present in the transmit FIFO
0000_0001	dma_tx_req is asserted when 1 or less data entry is present in the transmit FIFO
0000_0010	dma_tx_req is asserted when 2 or less data entries are present in the transmit FIFO
0000_0011	dma_tx_req is asserted when 3 or less data entries are present in the transmit FIFO
:	:
:	:
1111_1100	dma_tx_req is asserted when 252 or less data entries are present in the transmit FIFO
1111_1101	dma_tx_req is asserted when 253 or less data entries are present in the transmit FIFO
1111_1110	dma_tx_req is asserted when 254 or less data entries are present in the transmit FIFO
1111_1111	dma_tx_req is asserted when 255 or less data entries are present in the transmit FIFO

6.2.22 DMARDLR

- ❖ **Name:** DMA Receive Data Level
- ❖ **Size:** *RX_ABW*
- ❖ **Address Offset:** 0x54
- ❖ **Read/write access:** read/write

This register is only valid when DW_apb_ssi is configured with a set of DMA interface signals (SSI_HAS_DMA = 1). When DW_apb_ssi is not configured for DMA operation, this register will not exist and writing to its address will have no effect; reading from its address will return zero.



Bits	Name	R/W	Description
RX_ABW-1:0	DMARDL	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 = DMARDL+1; that is, dma_rx_req is generated when the number of valid data entries in the receive FIFO is equal to or above this field value + 1, and RDMAE=1. Refer to Table 6-7 for the field decode. Reset Value: 0x0

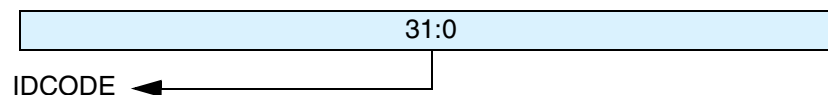
Table 6-7 DMARDL Decode Value

DMARDL Value	Description
0000_0000	dma_rx_req is asserted when 1 or more data entries are present in the receive FIFO
0000_0001	dma_rx_req is asserted when 2 or more data entries are present in the receive FIFO
0000_0010	dma_rx_req is asserted when 3 or more data entries are present in the receive FIFO
0000_0011	dma_rx_req is asserted when 4 or more valid data entries are present in the receive FIFO
:	:
:	:
1111_1100	dma_rx_req is asserted when 253 or more data entries are present in the receive FIFO
1111_1101	dma_rx_req is asserted when 254 or more data entries are present in the receive FIFO
1111_1110	dma_rx_req is asserted when 255 or more data entries are present in the receive FIFO
1111_1111	dma_rx_req is asserted when 256 data entries are present in the receive FIFO

6.2.23 IDR

- ❖ **Name:** Identification Register
- ❖ **Size:** 32 bits
- ❖ **Address Offset:** 0x58
- ❖ **Read/write access:** read-only

This read-only register is available for use to store a peripheral identification code.

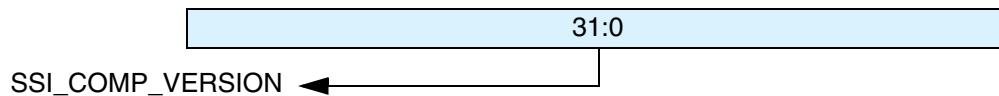


Bits	Name	R/W	Description
31:0	IDCODE	R	Identification Code. This register contains the peripherals identification code, which is written into the register at configuration time using coreConsultant. Reset Value: N/A

6.2.24 SSI_COMP_VERSION

- ❖ **Name:** coreKit version ID register
- ❖ **Size:** 32 bits
- ❖ **Address Offset:** 0x5C
- ❖ **Read/write access:** read-only

This read-only register stores the specific DW_apb_ssi component version.



Bits	Name	R/W	Description
31:0	SSI_COMP_VERSION	R	<p>Contains the hex representation of the Synopsys component version. Consists of ASCII value for each number in the version, followed by *. For example 32_30_31_2A represents the version 2.01*.</p> <p>Reset Value: See the releases table in the AMBA 2 release notes</p>

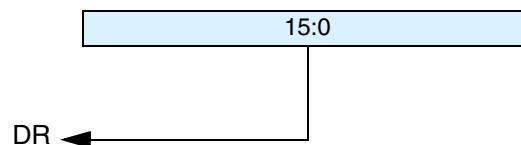
6.2.25 DR

- ❖ **Name:** Data Register
- ❖ **Size:** 16 bits
- ❖ **Address Offset:** 0x60 to 0xec
- ❖ **Read/write access:** read/write

The DW_apb_ssi data register is a 16-bit read/write buffer for the transmit/receive FIFOs. When the register is read, data in the receive FIFO buffer is accessed. When it is written to, data are moved into the transmit FIFO buffer; a write can occur only when SSI_EN = 1. FIFOs are reset when SSI_EN = 0.



Note The **DR** register in the DW_apb_ssi occupies thirty-six 32-bit address locations of the memory map to facilitate AHB burst transfers. Writing to any of these address locations has the same effect as pushing the data from the pwwdata bus into the transmit FIFO. Reading from any of these locations has the same effect as popping data from the receive FIFO onto the prdata bus. The FIFO buffers on the DW_apb_ssi are not addressable.



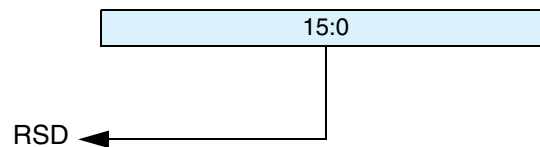
Bits	Name	R/W	Description
15:0	DR	R/W	Data Register. When writing to this register, you must right-justify the data. Read data are automatically right-justified. Read = Receive FIFO buffer Write = Transmit FIFO buffer Reset Value: 0x0

6.2.26 RX_SAMPLE_DLY

- ❖ **Name:** Rx Sample Delay Register
- ❖ **Size:** 8 bits
- ❖ **Address Offset:** 0xfc
- ❖ **Read/write access:** read/write

This register is valid only when the DW_apb_ssi is configured with rxd (Receive Data) sample delay logic; that is, when SSI_HAS_RX_SAMPLE_DELAY = 1. When the DW_apb_ssi is not configured with rxd sample delay logic, this register does not exist and writing to its address location has no effect; reading from its address returns zero (0).

This register controls the number of ssi_clk cycles that are delayed – from the default sample time – before the actual sample of the rxd input signal occurs. It is impossible to write to this register when the DW_apb_ssi is enabled; the DW_apb_ssi is enabled and disabled by writing to the SSIENR register.

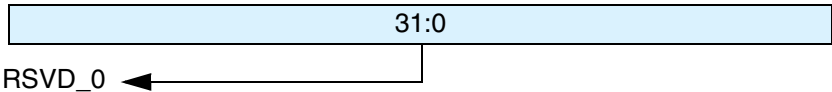


Bits	Name	R/W	Description
7:0	RSD	R/W	Receive Data (rxd) Sample Delay. This register is used to delay the sample of the rxd input signal. Each value represents a single ssi_clk delay on the sample of the rxd signal. NOTE: If this register is programmed with a value that exceeds the depth of the internal shift registers (SSI_RX_DLY_SR_DEPTH), a zero (0) delay will be applied to the rxd sample. Reset Value: 0x0

6.2.27 **RSVD_0**

- ❖ **Name:** Reserved location for future use
- ❖ **Size:** 32 bits
- ❖ **Address Offset:** 0xf4
- ❖ **Read/write access:** N/A

This register is reserved for future use.

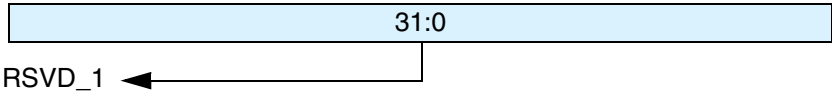


Bits	Name	R/W	Description
31:0	N/A	NA/	N/A Reset Value: N/A

6.2.28 **RSVD_1**

- ❖ **Name:** Reserved location for future use
- ❖ **Size:** 32 bits
- ❖ **Address Offset:** 0xf8
- ❖ **Read/write access:** N/A

This register is reserved for future use.

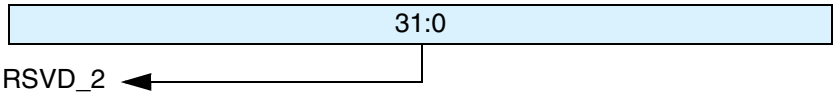


Bits	Name	R/W	Description
31:0	N/A	NA/	N/A Reset Value: N/A

6.2.29 RSVD_2

- ❖ **Name:** Reserved location for future use
- ❖ **Size:** 32 bits
- ❖ **Address Offset:** 0xfc
- ❖ **Read/write access:** N/A

This register is reserved for future use.



Bits	Name	R/W	Description
31:0	N/A	NA/	N/A Reset Value: N/A

7

Programming the DW_apb_ssi

This chapter describes the programmable features of the DW_apb_ssi.

7.1 Programming Considerations

You should program the following features during the configuration setup:

- ❖ APB data bus width
- ❖ Type of device configuration; that is, serial master or serial slave
- ❖ Depth of receive and transmit FIFO buffers
- ❖ Peripheral ID code
- ❖ Whether to include DMA handshaking interface signals
- ❖ Whether the interrupt level is active high or active low
- ❖ Whether interrupts are individual or combined
- ❖ Whether to generate a clock enable input for the ssi_clk
- ❖ Whether or not pclk and ssi_clk are synchronous
- ❖ Whether to hardcode the frame format, and what type of frame format:
 - ◆ Motorola SPI – requires setting the serial clock polarity and phase
 - ◆ Texas Instruments Synchronous Serial Protocol
 - ◆ National Semiconductor Microwire

8

Verification

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

**Note**

The DW_apb_ssi verification testbench is built with DesignWare Verification IP (VIP). Please make sure you have the supported version of the VIP components for this release, otherwise, you may experience some tool compatibility problems. For more information about supported tools in this release, refer to the [DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide](#).

8.1 Overview of Vera Tests

The DW_apb_ssi verification testbench performs the set of tests in this section, which have been written to exhaustively verify the functionality and have also achieved maximum RTL code coverage. All tests use the APB Interface to dynamically program memory-mapped registers during tests.

8.1.1 APB Interface

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

- ❖ All address locations are written to with valid data
- ❖ Configured MacroCell is AMBA-compliant
- ❖ Read/write coherent
- ❖ Reset value of all registers
- ❖ Functionality of all registers

**Note**

DW_apb_ssi does not start any operation when SSI_EN is held low. This control bit is verified separately. Regardless of programmed values of the registers, there is no activity on the component interface or within DW_apb_ssi once bit 0 of SSI Enable Register (**SSIENR**) is 0. Within the HDL code, there are a number of checkers that insert error information into the log file when the simulation is performed on an invalid configuration.

8.1.2 DW_apb_ssi as Master

This suite of tests is run only when the DW_apb_ssi is configured as a master, during which time all transfers are initiated by the DW_apb_ssi. When a master, the DW_apb_ssi must generate the clock `clk_out`.

8.1.3 DW_apb_ssi as Slave

This suite of tests is run only when the DW_apb_ssi is configured as a slave. Similar to the tests developed for the master, the driving force is the serial-master BFM. The DW_apb_ssi should be tested for all frame formats, for all transfer modes, for all length of data frames, for all combinations of clock polarity, and clock phase. The serial-master BFM initiates all the serial transfers.

8.1.4 DW_apb_ssi with DMA Interface

This suite of tests is only run when the DW_apb_ssi is configured with a handshaking interface. DMA is implemented for both Transmit and Receive. These test verify the following:

- ❖ SSI_HAS_DMA
- ❖ After reset that all DMA control registers read zero.
- ❖ It is possible to set the Transmit/Receive DMA enable bit via an APB write. Confirm it is possible to clear these bits via an APB write.¹⁷
- ❖ Once the transmit enable bit is set, a DMA transmit transfer request is generated provided the number of entries in the FIFO is less than or equal to the DMA Data Level. The DMAing BFM is configured initially not to respond to requests. APB transfers can fill the FIFO and confirm that the request line will be removed as the level in the FIFO fills. Confirm that when the DMAing BFM is configured to respond that it fills the FIFO with sufficient data to remove the request.
- ❖ The `dma_tx_req` signal is active when the number of entries in the transmit FIFO is equal to or below the DMA Data Level. And remains active.
- ❖ The `dma_tx_single` signal is asserted when there is at least one empty location in the transmit FIFO. Confirm that it is cleared when the transmit FIFO is full.
- ❖ The `dma_tx_ack` signal causes a pulse on `dma_tx_req` so that the DMAing BFM can respond to requests for subsequent DMA transfers. The BFM only responds to rising edges on the request line.
- ❖ The `dma_tx_finish` signal clears the transmit enable bit. Confirm that the `dma_tx_req` and `dma_tx_single` lines are cleared. Confirm that `dma_tx_finish` stays active until `dma_tx_req` is sampled low.¹⁸
- ❖ All data words are transmitted via DMA and that there is never any data left within the DMAing BFM, one should be able to configure the BFM to read in bursts or in singles. This should be confirmed for a range of threshold values.
- ❖ The same operation exists for receive FIFOs.

Once a `dma_[r | t]x_finish` is received, the corresponding enable bit is cleared. Confirm that there are no subsequent requests for DMA once the enable is removed.

8.1.5 Interrupts

These tests verify the following:

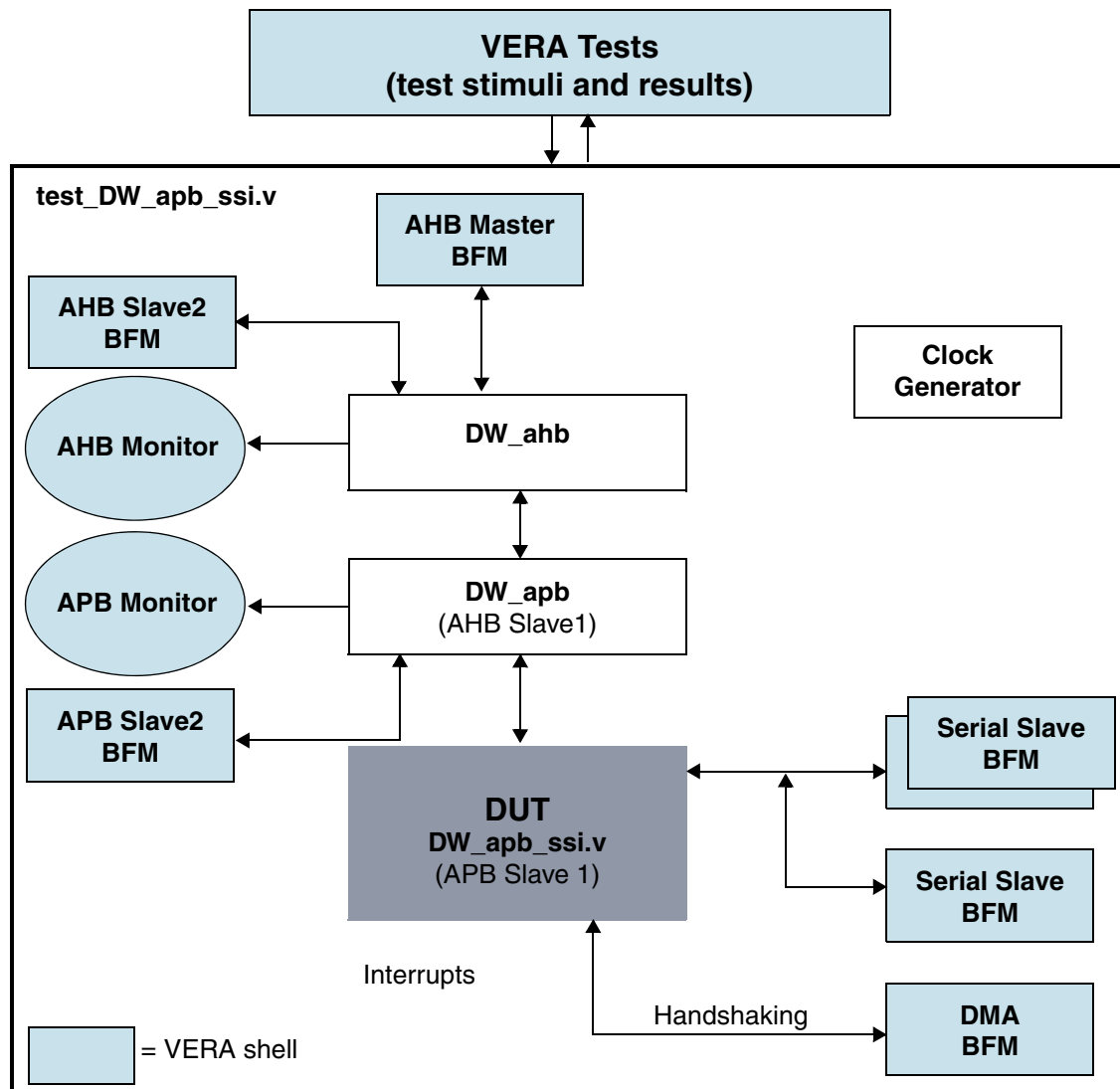
- ❖ The Transmit FIFO Empty interrupt is not active when the DW_apb_ssi is returned from reset.
- ❖ The Receive FIFO Full interrupt is not active when the DW_apb_ssi is returned from reset.
- ❖ The Transmit FIFO Overflow interrupt is not active when the DW_apb_ssi is returned from reset.
- ❖ The Receive FIFO Overflow interrupt is not active when the DW_apb_ssi is returned from reset.
- ❖ The Receive FIFO Underflow interrupt is not active when the DW_apb_ssi is returned from reset.
- ❖ The MultiMaster Contention interrupt is not active when the DW_apb_ssi is returned from reset.

8.2 Overview of DW_apb_ssi Testbench

As illustrated in [Figure 8-1](#) on page 126, the DW_apb_ssi testbench is a Verilog testbench that includes an instantiation 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 BFMs, a master slave BFM, and a DMA BFM to simulate and stimulate traffic to and from the DW_apb_ssi.

The test_DW_apb_ssi.v file shows the instantiation of the top-level MacroCell in a testbench and resides in the *workspace/src* directory. The testbench checks your configuration selected in the Specify Configuration task of coreConsultant. The testbench also determines if 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_ssi* form the actual testbench for DW_apb_ssi.

Figure 8-1 DW_apb_ssi Testbench



9

Integration Considerations

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

9.1 Reading and Writing from an APB Slave

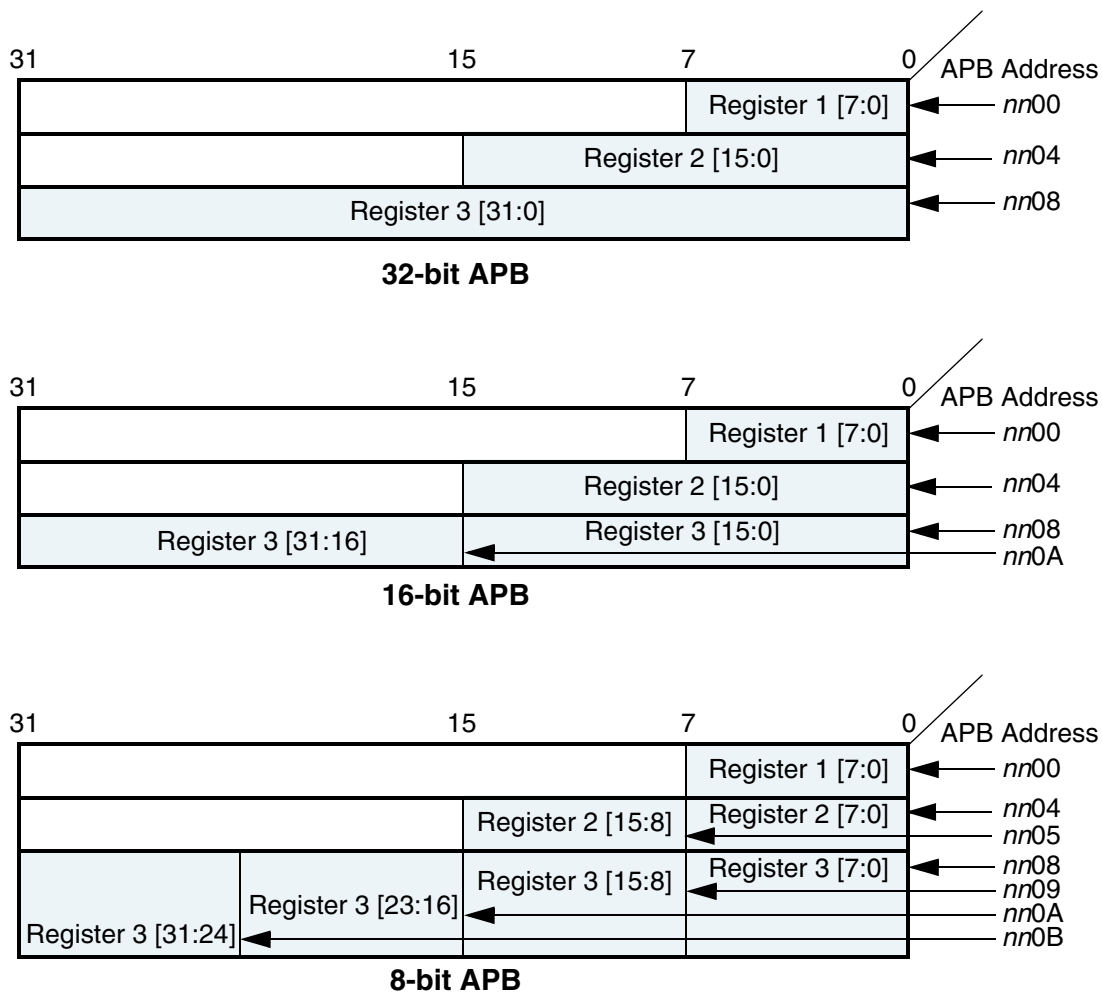
When writing to and reading from DesignWare APB slaves, you should consider the following:

- ❖ 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.

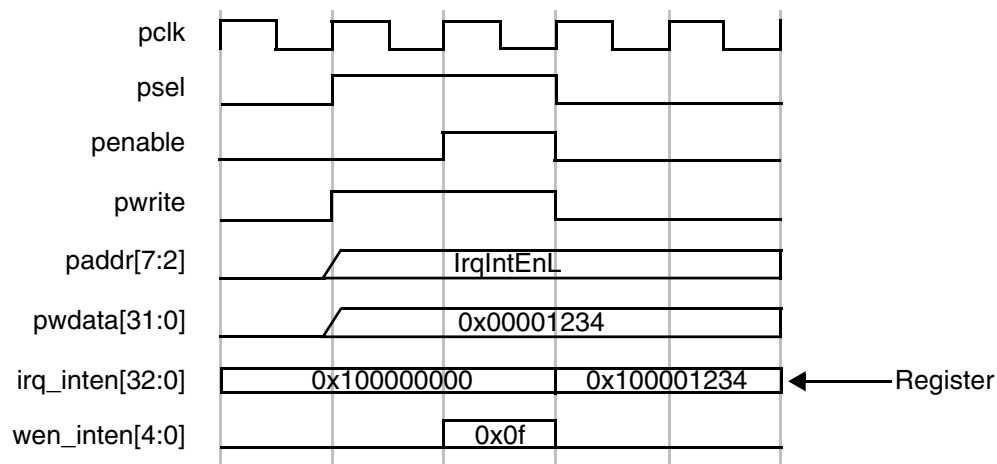
3. The register to be written to or read from is >16 and ≤ 32 bits

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

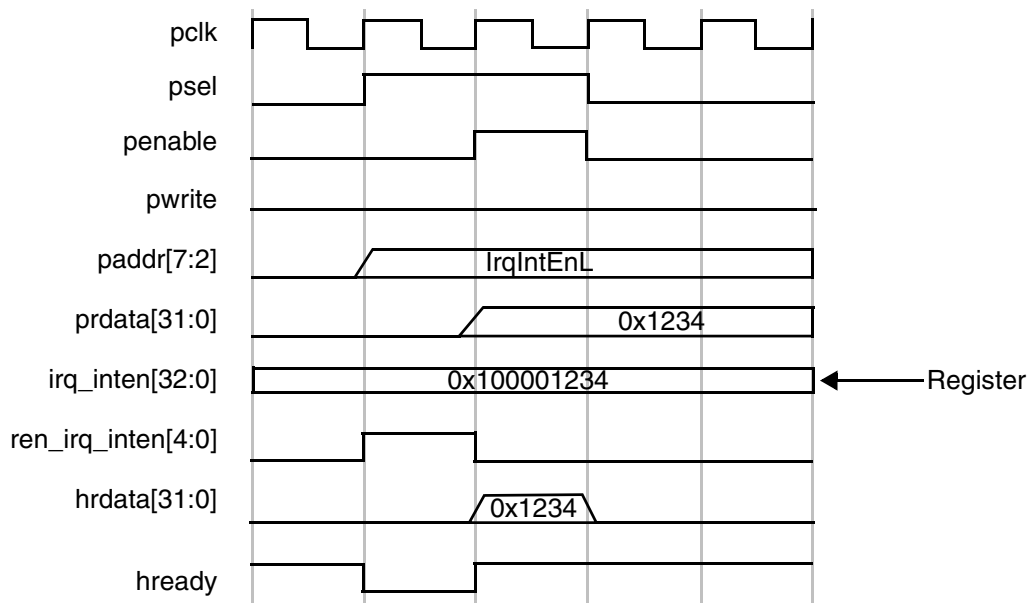
A write can occur after the first phase with **penable** low, or after the second phase when **penable** is high. The second phase is preferred and is used in all APB slave components. The timing diagram is shown with the write occurring after the second phase. Whenever the address on **paddr** matches 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 Coherency

Coherency is where bits within a register are logically connected. For instance, part of a register is read at time 1 and another part is read at time 2. Being coherent means that the part read at time 2 is at the same value it was when the register was read at time 1. The unread part is stored into a shadow register and this is read at time 2. When there is no coherency, no shadow registers are involved.

A bus master may need to be able to read the contents of a register, regardless of the data bus width, and be guaranteed of the coherency of the value read. A bus master may need to be able to write a register coherently regardless of the data bus width and use that register only when it has been fully programmed. This may need to be the case regardless of the relationship between the clocks.

Coherency enables a value to be read that is an accurate reflection of the state of the counter, independent of the data bus width, the counter width, and even the relationship between the clocks. Additionally, a value written in one domain is transferred to another domain in a seamless and coherent fashion.

Throughout this appendix the following terms are used:

- ❖ **Writing.** A bus master programs a configuration register. An example is programming the load value of a counter into a register.
- ❖ **Transferring.** The programmed register is in a different clock domain to where it is used, therefore, it needs to be transferred to the other clock domain.
- ❖ **Loading.** Once the programmed register is transferred into the correct clock domain, it needs to be loaded or used to perform its function. For example, once the load value is transferred into the counter domain, it gets loaded into the counter.

9.5.1 Writing Coherently

Writing coherently means that all the bits of a register can be written at the same time. A peripheral may have programmable registers that are wider than the width of the connected APB data bus, which prevents all the bits being programmed at the same time unless additional coherency circuitry is provided.

The programmable register could be the load value for a counter that may exist in a different clock domain. Not only does the value to be programmed need to be coherent, it also needs to be transferred to a different clock domain and then loaded into the counter. Depending on the function of the programmable register, a qualifier may need to be generated with the data so that it knows when the new value is currently transferred and when it should be loaded into the counter.

Depending on the system and on the register being programmed, there may be no need for any special coherency circuitry. One example that requires coherency circuitry is a 32-bit timer within an 8-bit APB system. The value is entirely programmed only after four 8-bit wide write transfers. It is safe to transfer or use the register when the last byte is currently written. An example where no coherency is required is a 16-bit wide timer within a 16-bit APB system. The value is entirely programmed after a single 16-bit wide write transfer.

Coherency circuitry enables the value to be loaded into the counter only when fully programmed and crossed over clock domains if the peripheral clock is not synchronous to the processor clock. While the load register is being programmed, the counter has access to the previous load value in case it needs to reload the counter.

Coherency circuitry is only added in cores where it is needed. The coherency circuitry incorporates an upper byte method that requires users to program the load register in LSB to MSB order when the peripheral width is smaller than the register width. When the upper byte is programmed, the value can be transferred and loaded into the load register. When the lower bytes are being programmed, they need to be stored in shadow registers so that the previous load register is available to the counter if it needs to reload.

When the upper byte is programmed, the contents of the shadow registers and the upper byte are loaded into the load register.

The upper byte is the top byte of a register. A register can be transferred and loaded into the counter only when it has been fully programmed. A new value is available to the counter once this upper byte is written into the register. The following table gives the relationship between the register width and the peripheral bus width for the generation of the correct upper byte. The numbers in the table represent bytes, Byte 0 is the LSB and Byte 3 is the MSB. NCR means that no coherency circuitry is required, as the entire register is written with one access.

Table 9-1 Upper Byte Generation

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

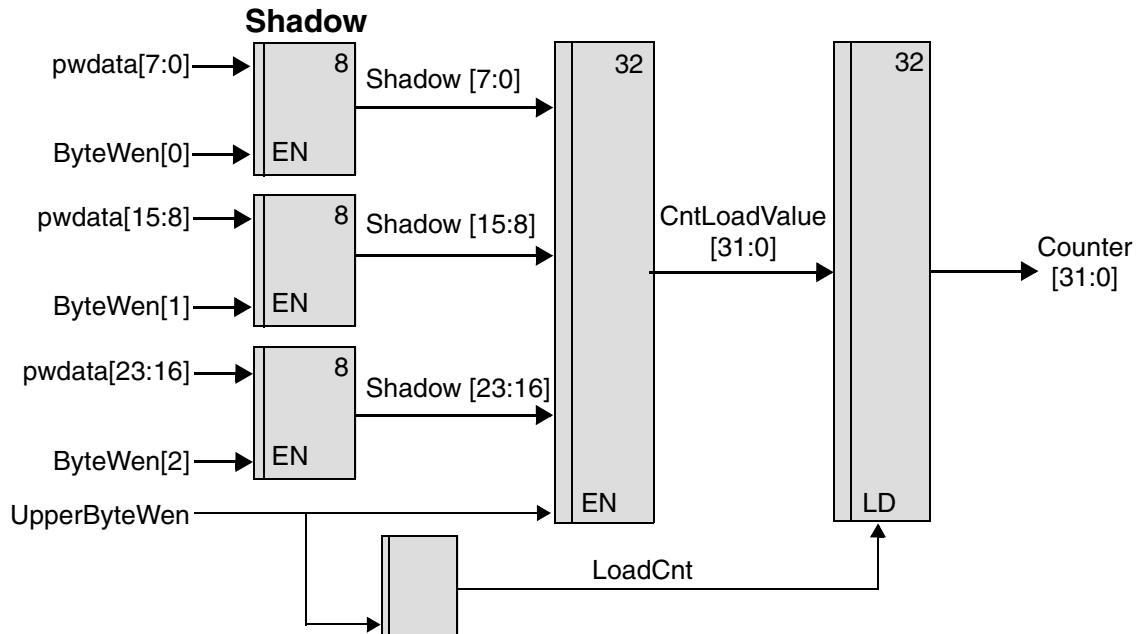
There are three relationship cases to be considered for the processor and peripheral clocks:

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

9.5.1.1 Identical Clocks

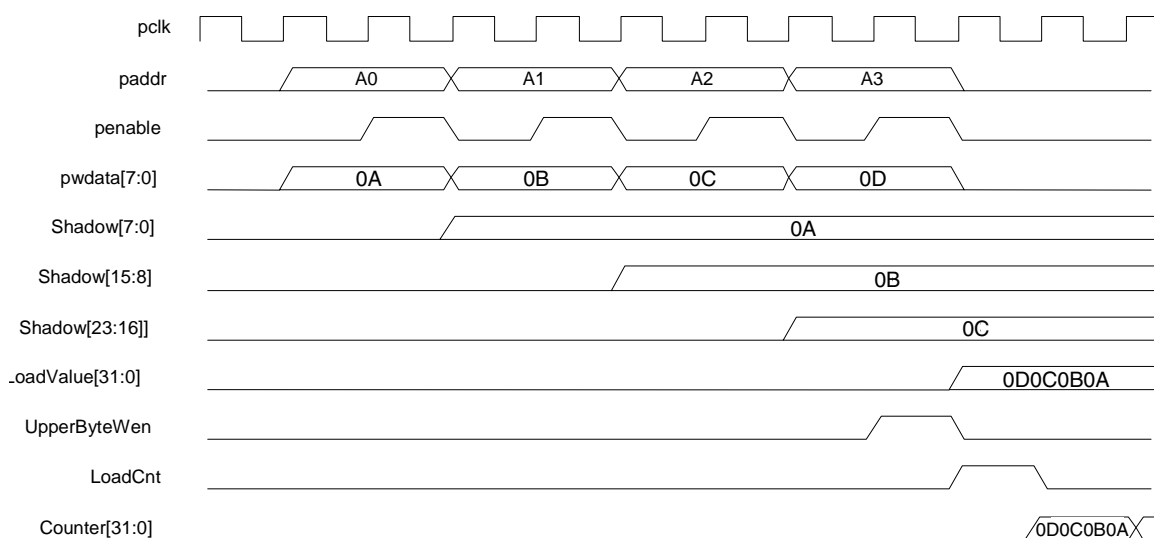
The following figure illustrates an RTL diagram for the circuitry required to implement the coherent write transaction when the APB bus clock and peripheral clocks are identical.

Figure 9-4 Coherent Loading – Identical Synchronous Clocks



The following timing diagram shows the shadow registers being loaded and then loaded into the counter when fully programmed. The LoadCnt signal lasts for one cycle and is used to load the counter with CntLoadValue.

Figure 9-5 Coherent Loading – Identical Synchronous Clocks



Each of the bytes that make up the load register are stored into shadow registers until the final byte is written. The shadow register is up to three bytes wide. The contents of the shadow registers and the final

byte are transferred into the CntLoadValue register when the final byte is written. The counter uses this register to load/initialize itself. If the counter is operating in a periodic mode, it reloads from this register each time the count expires.

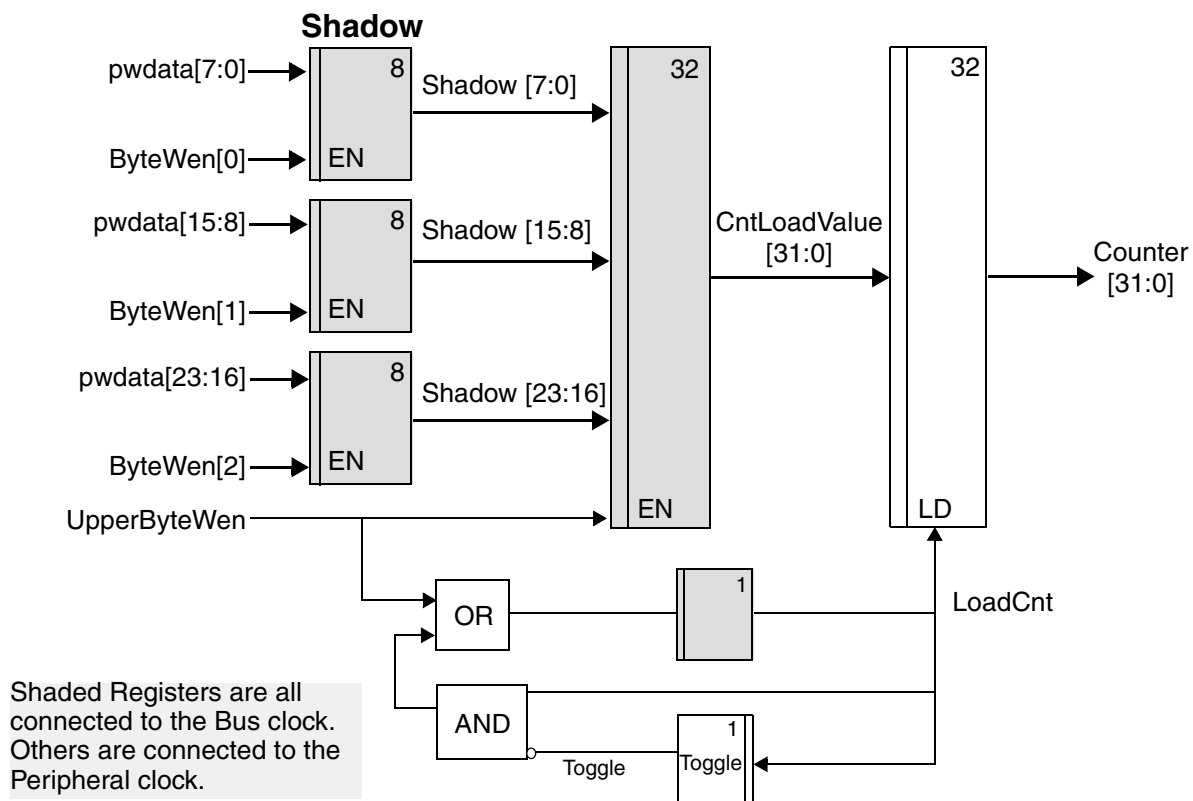
By using the shadow registers, the CntLoadValue is kept stable until it can be changed in one cycle. This allows the counter to be loaded in one access and the state of the counter is not affected by the latency in programming it. When there is a new value to be loaded into the counter initially, this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

9.5.1.2 Synchronous Clocks

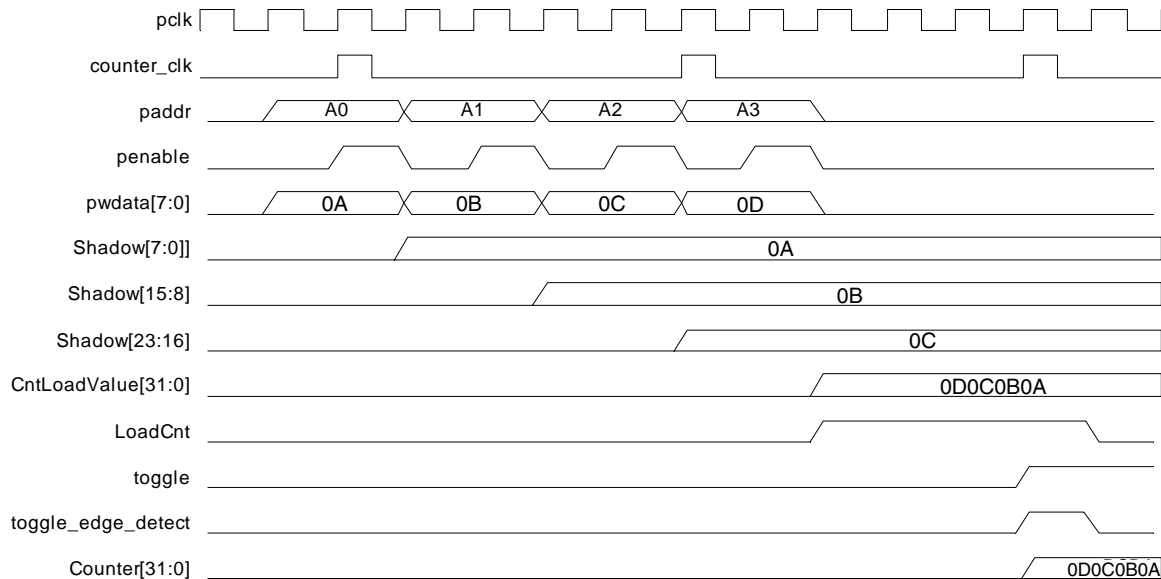
When the clocks are synchronous but do not have identical periods, the circuitry needs to be extended so that the LoadCnt signal is kept high until a rising edge of the counter clock occurs. This extension is necessary so that the value can be loaded, using LoadCnt, into the counter on the first counter clock edge. At the rising edge of the counter clock if LoadCnt is high, then a register clocked with the counter clock toggles, otherwise it keeps its current value. A circuit detecting the toggling is used to clear the original LoadCnt by looking for edge changes. The value is loaded into the counter when a toggle has been detected. Once it is loaded, the counter should be free to increment or decrement by normal rules.

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

Figure 9-6 Coherent Loading – Synchronous Clocks

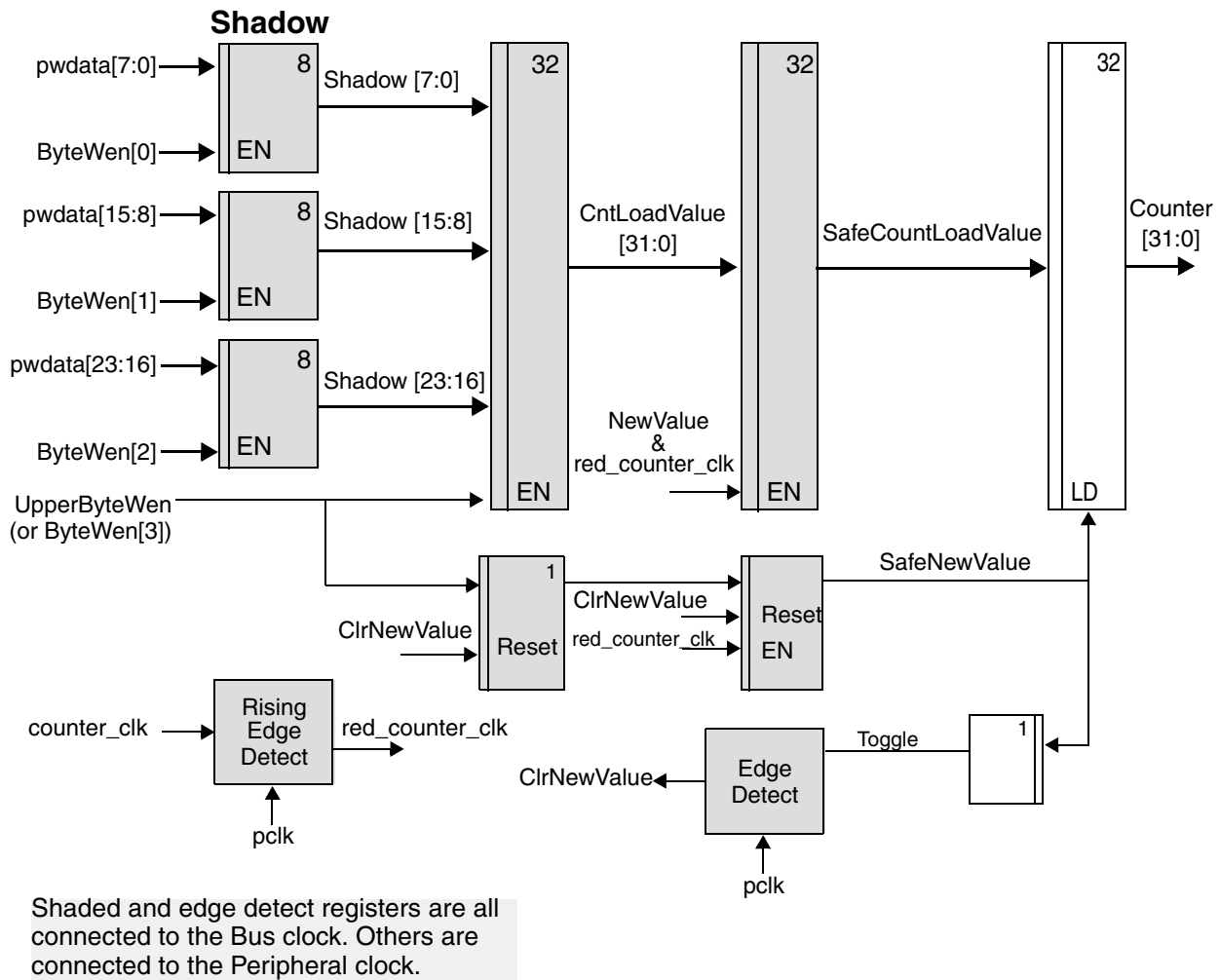


The following timing diagram shows the shadow registers being loaded and then loaded into the counter when fully programmed. The LoadCnt signal is extended until a change in the toggle is detected and is used to load the counter.

Figure 9-7 Coherent Loading – Synchronous Clocks

9.5.1.3 Asynchronous Clocks

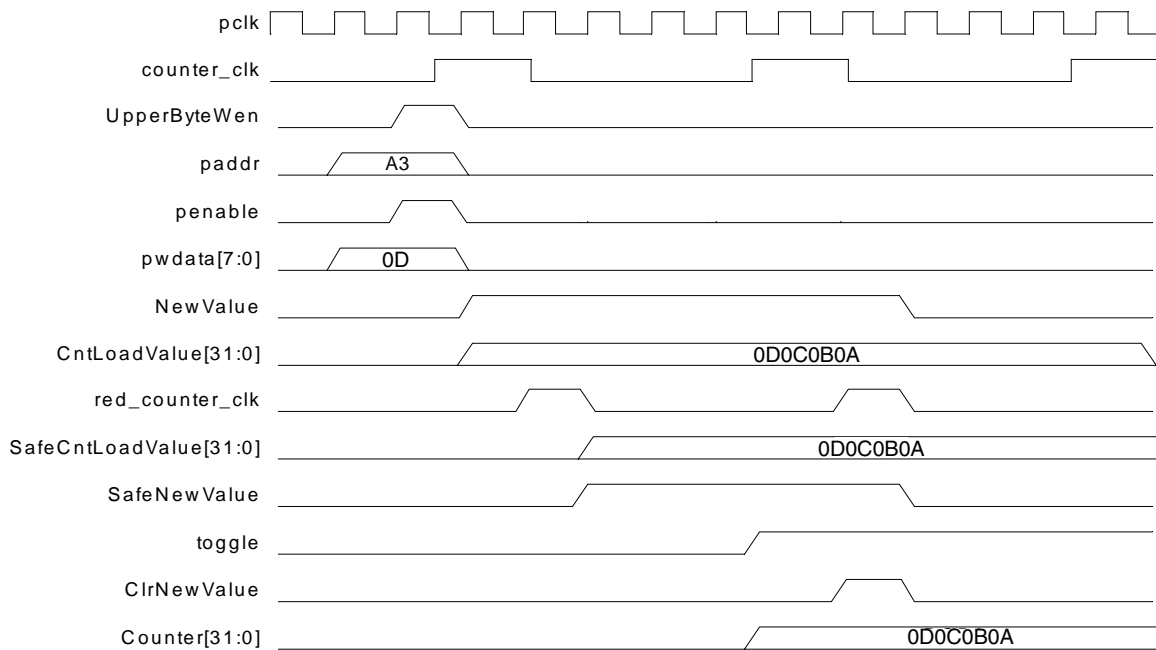
When the clocks are asynchronous, the processor clock needs to be three-times the speed of the peripheral clock for the re-timing to operate correctly. The high pulse time of the peripheral clock needs to be greater than the period of the processor clock. The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are asynchronous.

Figure 9-8 Coherent Loading – Asynchronous Clocks

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

Every time there is a rising edge of the peripheral clock detected, the CntLoadValue is transferred into a SafeCntLoadValue. This value is used to transfer the load value across the clock domains. The SafeCntLoadValue only changes a number of bus clock cycles after the peripheral clock edge changes. A counter running on the peripheral clock is able to use this value safely. It could be up to two peripheral clock periods before the value is loaded into the counter. Along with this loaded value, there also is a single bit transferred that is used to qualify the loading of the value into the counter.

The following timing diagram does not show the shadow registers being loaded. This is identical to the loading for the other clock modes. The NewValue signal is extended until a change in the toggle is detected and is used to update the safe value. The SafeNewValue is used to load the counter at the rising edge of the peripheral clock. Each time a new value is written the toggle bit is flipped and the edge detection of the toggle is used to remove both the NewValue and the SafeNewValue.

Figure 9-9 Coherent Loading – Asynchronous Clocks

9.5.2 Reading Coherently

For writing to registers, an upper-byte concept is proposed for solving coherency issues. For read transactions, a lower-byte concept is required. The following table provides the relationship between the register width and the bus width for the generation of the correct lower byte. Depending on the bus width and the register width, there may be no need to save the upper bits because the entire register is read in one access, in which case there is no problem with coherency. When the lower byte is read, the remaining upper bytes within the counter register are transferred into a holding register. The holding register is the source for the remaining upper bytes. Users must read LSB to MSB for this solution to operate correctly. NCR means that no coherency circuitry is required, as the entire register is read with one access.

Table 9-2 Lower Byte Generation

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

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

- ❖ Identical and/or synchronous
- ❖ Asynchronous

9.5.2.1 Synchronous Clocks

When the clocks are identical and/or synchronous, the remaining unread bits (if any) need to be saved into a holding register once a read is started. The first read byte must be the lower byte provided in the previous table, which causes the other bits to be moved into the holding register, *SafeCntVal*, provided that the register cannot be read in one access. The upper bytes of the register are read from the holding register rather than the actual register so that the value read is coherent. This is illustrated in the following figure and timing diagram.

Figure 9-10 Coherent Registering – Synchronous Clocks

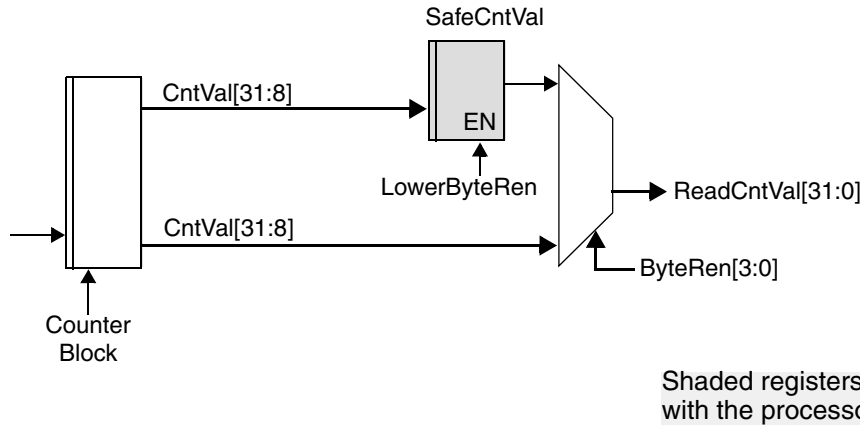
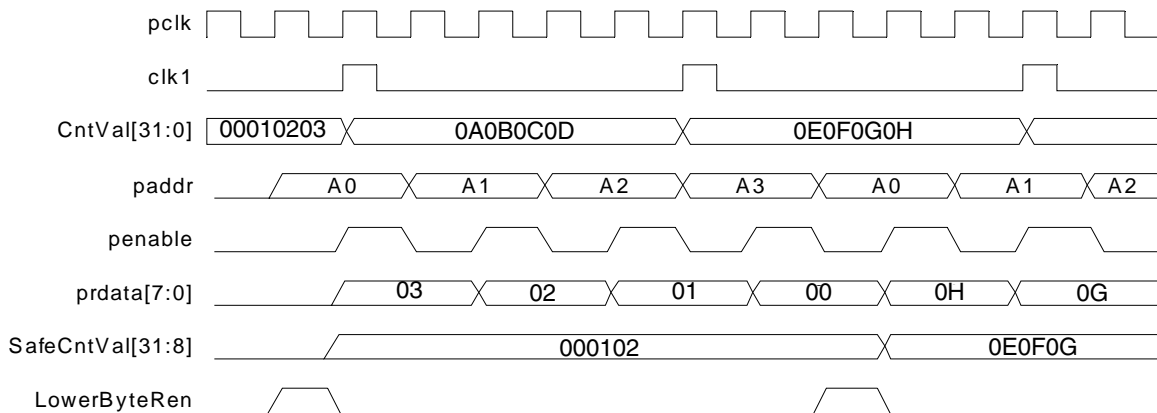


Figure 9-11 Coherent Registering – Synchronous Clocks



9.5.2.2 Asynchronous Clocks

When the clocks are asynchronous, the processor clock needs to be three times the speed of the peripheral clock for the re-timing to operate correctly. The high pulse time of the peripheral clock needs to be greater than the period of the processor clock.

To safely transfer a counter value from the counter clock domain to the bus clock domain, the counter clock signal should be transferred to the bus clock domain. When the rising edge detect of this re-timed counter clock signal is detected, it is safe to use the counter value to update a shadow register that holds the current value of the counter.

While reading the counter contents it may take multiple APB transfers to read the value.



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

Once a read transaction has started, the value of the upper register bits need to be stored into a shadow register so that they can be read with subsequent read accesses. Storing these upper bits preserves the coherency of the value that is being read. When the processor reads the current value it actually reads the contents of the shadow register instead of the actual counter value. The holding register is read when the bus width is narrower than the counter width. When the LSB is read, the value comes from the shadow register; when the remaining bytes are read they come from the holding register. If the data bus width is wide enough to read the counter in one access, then the holding registers do not exist.

The counter clock is registered and successively pipelined to sense a rising edge on the counter clock. Having detected the rising edge, the value from the counter is known to be stable and can be transferred into the shadow register. The coherency of the counter value is maintained before it is transferred, because the value is stable.

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

Figure 9-12 Coherency and Shadow Registering – Asynchronous Clocks

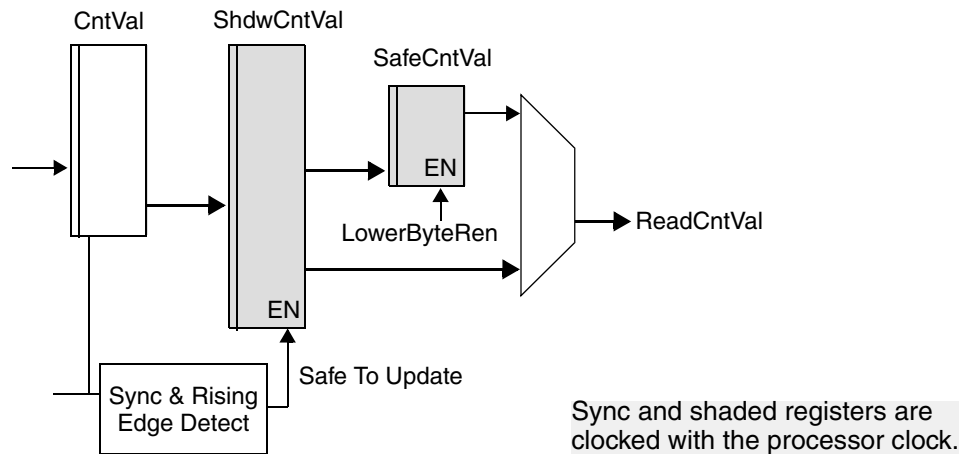
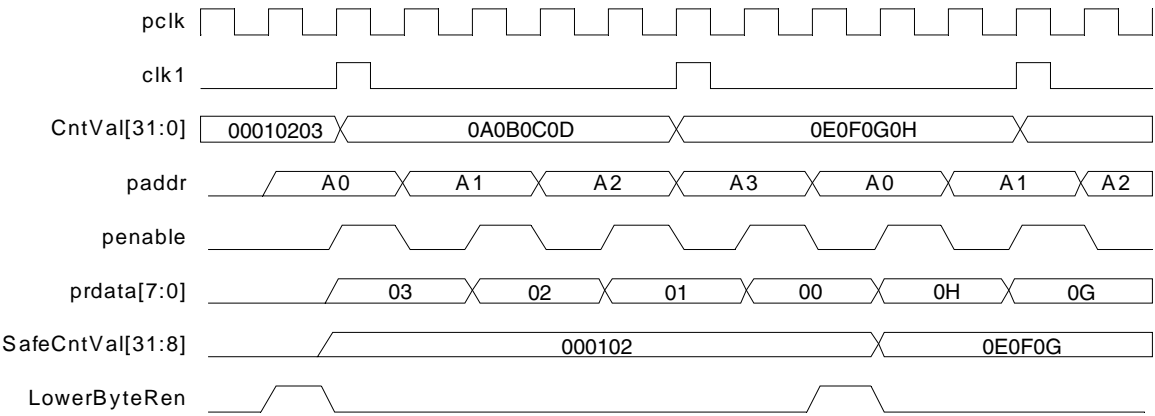


Figure 9-13 Transfer to Shadowing Registers– Asynchronous Clocks



A

Application Notes

This appendix contains useful “Application Note” information that may be helpful to you in using the DW_apb_ssi component.

A.1 Interfacing DW_apb_ssi and Atmel SPI Devices

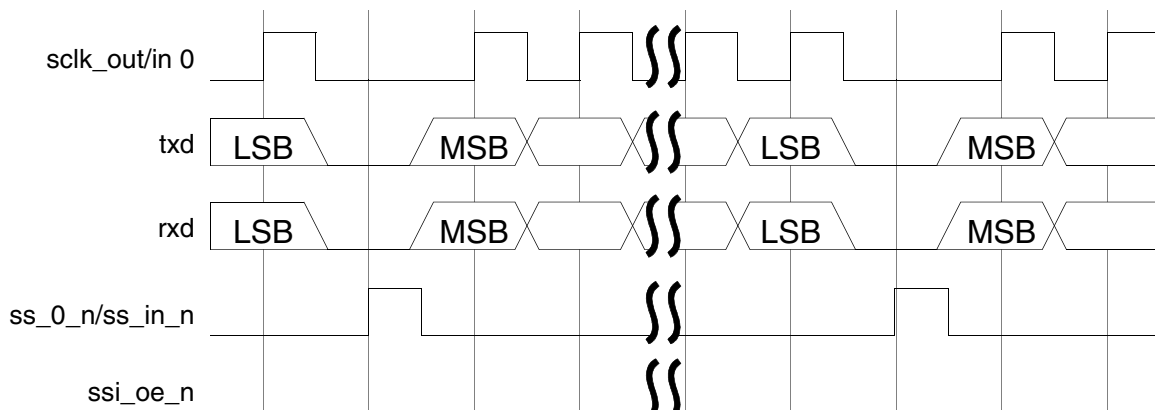
Synopsys and Atmel have taken different interpretations of the Motorola SPI serial protocol from the specification. The DW_apb_ssi component is able to communicate with Atmel SPI peripherals but must be programmed slightly differently than with other vendor devices.

The differences between the two devices concern the serial clock phase (SCPH) and serial clock polarity (SCPOL) configuration parameters for the SPI protocol.

A.1.1 Synopsys SPI Operation

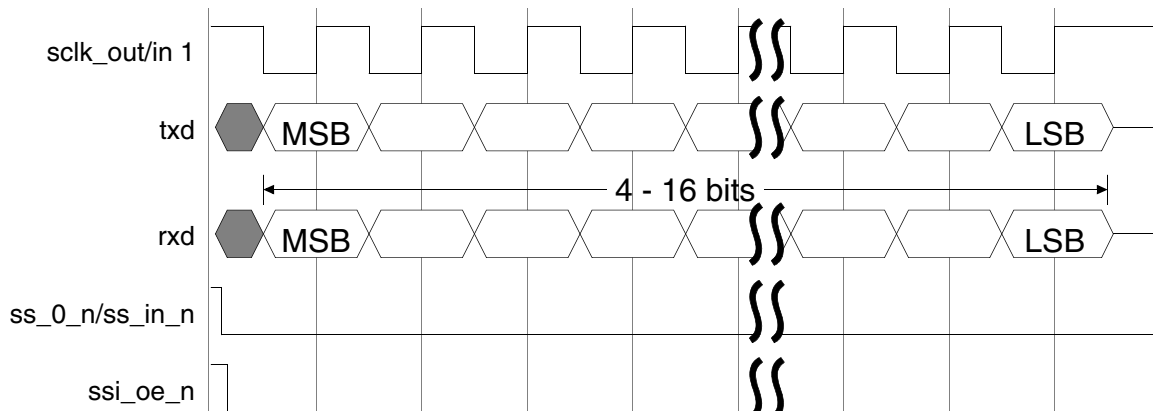
When the default serial clock phase is set to logic ‘0’ (SCPH = 0) and the default serial clock polarity is set to logic ‘0’ (SCPOL = 0), the DW_apb_ssi master device toggles the slave select output (ss_0_n) before beginning each new data frame of a continuous serial transfer. This occurs because data transmission starts on the falling edge of the slave select signal. Data is propagated on the negative edge of the serial clock and captured on the positive edge of the serial clock. The inactive state of the serial clock in this mode is logic ‘0’. [Figure A-1](#) shows a continuous transfer from a DW_apb_ssi master with the SCPH and SCPOL configuration parameters both set to logic ‘0’.

Figure A-1 DW_apb_ssi SPI: Continuous Transfer where SCPH = 0 and SCPOL = 0



When both the serial clock phase (SCPH) and the serial clock polarity (SCPOL) configuration parameters are set to logic '1', the DW_apb_ssi master device transmits/captures the most significant bit (MSB) of the new data frame directly after the least significant bit (LSB) from the previous data frame. The slave select signal remains active for the duration of the serial transfer. This occurs because data transmission does not begin until the first serial clock edge after the slave select signal is active. Data is propagated on the falling edge of the serial clock and captured on the rising edge of the serial clock. The inactive state of the serial clock in this mode is logic '1'. [Figure A-2](#) shows a continuous transfer from a DW_apb_ssi master with SCPH = 1 and SCPOL = 1.

Figure A-2 DW_apb_ssi SPI: Continuous Transfer where SCPH=1 and SCPOL=1



A.1.2 Atmel SPI Operation

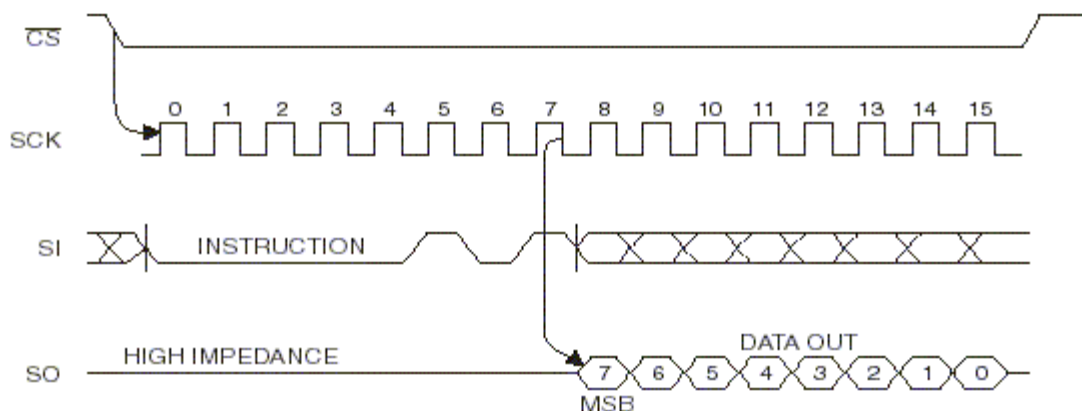
When the Atmel peripheral advertises both the serial clock phase (SCPH) and the serial clock polarity (SCPOL) configuration parameters set to logic '0', it behaves almost exactly like the DW_apb_ssi component when programmed with SCPH = 1 and SCPOL = 1.

The Atmel device transmits/captures the MSB of the new data frame directly after the LSB from the previous data frame. The slave select signal remains active for the duration of the serial transfer. Data is propagated on the falling edge of the serial clock and captured on the rising edge of the serial clock. This behavior matches exactly the behavior of the DW_apb_ssi device when programmed with SCPH and SCPOL set to logic '1'.

A timing diagram from an Atmel data sheet (AT25F4096), shown [Figure A-3](#), illustrates a continuous SPI transfer to one of their SPI memory devices with both SCPH and SCPOL set to logic '0'. The only visible

difference between the Atmel timing diagram and the DW_apb_ssi timing diagram (SCPH = 1, SCPOL = 1) is the inactive level of the serial clock.

Figure A-3 Atmel SPI: Continuous Transfer with SCPH=0 and SCPOL=0



A.1.3 Interoperability between DW_apb_ssi and Atmel Devices

In order for the DW_apb_ssi component to communicate with an Atmel peripheral, you must invert the logic on the advertised SCPH and SCPOL parameters when programming the DW_apb_ssi component.

B

Glossary

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

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

DesignWare cores	A specific collection of synthesizable cores that are licensed individually. For more information, refer to www.synopsys.com/designware .
DesignWare Library	A collection of synthesizable IP and verification IP components that is authorized by a single DesignWare license. Products include SmartModels, VMT model suites, DesignWare Memory Models, Building Block IP, and the DesignWare Synthesizable Components.
dual role device	Device having the capabilities of function and host (limited).
endian	Ordering of bytes in a multi-byte word; see also little-endian and big-endian.
Full-Functional Mode	A simulation model that describes the complete range of device behavior, including code execution. See also BFM.
GPIO	General Purpose Input Output.
GTECH	A generic technology view used for RTL simulation of encrypted source code by non-Synopsys simulators.
hard IP	Non-synthesizable implementation IP.
HDL	Hardware Description Language – examples include Verilog and VHDL.
IIP	Implementation Intellectual Property — A generic term for synthesizable HDL and non-synthesizable “hard” IP in all of its forms (coreKit, component, core, MacroCell, and so on).
implementation view	The RTL for a core. You can simulate, synthesize, and implement this view of a core in a real chip.
instantiate	The act of placing a core or model into a design.
interface	Set of ports and parameters that defines a connection point to a component.
IP	Intellectual property — A term that encompasses simulation models and synthesizable blocks of HDL code.
little-endian	Data format in which the least-significant byte comes first.
MacroCell	Bigger IP blocks (6811, 8051, memory controller) available in the DesignWare Library and delivered with coreConsultant.
master	Device or model that initiates and controls another device or peripheral.
model	A Verification IP component or a Design View of a core.
monitor	A device or model that gathers performance statistics of a system.
non-blocking command	A testbench command that advances to the next testbench statement without waiting for the command to complete.
peripheral	Generally refers to a small core that has a bus connection, specifically an APB interface.

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

Index

A

active command queue
 definition [147](#)

activity
 definition [147](#)

AHB
 definition [147](#)

AMBA
 definition [147](#)

APB
 definition [147](#)

APB bridge
 definition [147](#)

APB Interface, and DW_apb_ssi [73](#)

APB_DATA_WIDTH [75](#)

application design
 definition [147](#)

arbiter
 definition [147](#)

B

Baud rate select register [102](#)

BAUDR [102](#)

BFM
 definition [147](#)

big-endian
 definition [147](#)

Block diagram, of DW_apb_ssi [13](#)

blocked command stream
 definition [147](#)

blocking command
 definition [147](#)

Buffers, transmit and receive FIFOs [32](#)

bus bridge
 definition [148](#)

C

Clock ratios [31](#)

Coherency
 about [132](#)
 read [138](#)
 write [132](#)

command channel
 definition [148](#)

command stream
 definition [148](#)

component
 definition [148](#)

configuration
 definition [148](#)

configuration intent
 definition [148](#)

Configuration parameters [75](#)

Control Register 0 (CTRLR0) [94](#)

Control Register 1 (CTRLR1) [98](#)

core
 definition [148](#)

core developer
 definition [148](#)

core integrator
 definition [148](#)

coreAssembler
 definition [148](#)
 overview of usage flow [21](#)

coreConsultant
 definition [148](#)
 overview of usage flow [18](#)

coreKit
 definition [148](#)

Customer Support [8](#)

cycle command
definition 148

D

Data register 117
decoder
definition 148
design context
definition 148
design creation
definition 148
Design View
definition 148
DesignWare cores
definition 149
DesignWare Library
definition 149
DesignWare Synthesizable Components
definition 148
DMA Controller
and DW_apb_ssi 65
DMA interface, signals 84
dma_rx_ack 85
dma_rx_req 85
dma_rx_single 85
dma_tx_ack 85
dma_tx_req 84
dma_tx_single 85
DR 117
dual role device
definition 149
DW_apb
slaves
read timing operation 130
write timing operation 129
DW_apb_ssi
block diagram of 13
clock ratios 31
description 29
functional description 29
functional description of 13
I/O description 81
interfacing to 81
Master mode 36

memory map of 91
parameters 75
programming of 91, 121
signal description 83
slave mode 46
testbench
overview of 125
overview of tests 123

E

EEPROM Read mode 35
endian
definition 149
Environment, licenses 15

F

Full-Functional Mode
definition 149
Functional description 13, 29
Functional description, of DW_apb_ssi 29

G

GPIO
definition 149
GTECH
definition 149

H

hard IP
definition 149
HDL
definition 149

I

I/O signals
description of 81
I/O signals, description of 83
ICR 113
Identification register 116
IDR 116
IIP
definition 149
implementation view
definition 149
IMR 108
instantiate
definition 149

interface
 definition 149

Interfaces
 APB 73
 DMA Controller 65

Interfacing
 to DW_apb_ssi 81

Interrupt Clear Register 113

Interrupt mask register 108

Interrupt Status Register 109

Interrupts, transmit and receive FIFOs 33

IP
 definition 149

ISR 109

L

Licenses 15

little-endian
 definition 149

M

MacroCell
 definition 149

master
 definition 149

Master mode 36

Master transfer flow 45

Memory map, of DW_apb_ssi 91

model
 definition 149

monitor
 definition 149

Motorola SPI, description of 49

MSTICR 113

Multi-master Interrupt Clear Register 113

N

National Semiconductor Microwire, description of 55

non-blocking command
 definition 149

O

Operation modes
 Master mode 36
 Slave mode 46

Output files

GTECH 26

RTL-level 25

Simulation model 26

synthesis 26

verification 27

P

paddr 84

Parameters, description of 75

pclk 83

penable 83

peripheral
 definition 149

prdata 84

presetn 83

Programming DW_apb_ssi
 memory map 91, 121
 registers 94

psel 83

pwdata 84

pwrite 84

R

Raw interrupt status register 110

Read coherency
 about 138
 and asynchronous clocks 139
 and synchronous clocks 139

Reading, from unused locations 127

Receive FIFO
 buffers 32
 interrupts 33

Receive FIFO level register 105

Receive FIFO Overflow Interrupt Clear Register 112

Receive FIFO threshold level register 104

Receive only mode 35

Register
 RXFLR 105

Registers
 BAUDR 102
 CTRLR0 94
 CTRLR1 98
 DR 117
 ICR 113

- IDR [116](#)
- IMR [108](#)
- ISR [109](#)
- MSTICR [113](#)
- RISR [110](#)
- RXFTLR [104](#)
- RXOICR [112](#)
- SER [101](#)
- SR [106](#)
- TXFLR [105](#)
- TXFTLR [103](#)
- TXOICR [111](#)
- RISR [110](#)
- RTL
 - definition [150](#)
- rx_d [86](#)
- RXFLR [105](#)
- RXFTLR [104](#)
- RXOICR [112](#)
- S**
- sclk_in [88](#)
- sclk_out [87](#)
- SDRAM
 - definition [150](#)
- SDRAM controller
 - definition [150](#)
- SER [101](#)
- Serial protocols
 - about [29](#)
 - Motorola SPI [49](#)
 - National Semiconductor Microwire [55](#)
 - Texas Instruments SSP [54](#)
- Signal description [83](#)
- Signals. *See* I/O signals
- Simulation
 - of DW_apb_ssi coreKit [125](#)
- slave
 - definition [150](#)
- Slave enable register [101](#)
- Slave mode [46](#)
- Slave transfer flow [48](#)
- SoC
 - definition [150](#)
- SoC Platform
 - AHB contained in [11](#)
 - APB, contained in [11](#)
 - defined [11](#)
- soft IP
 - definition [150](#)
- SR [106](#)
- ss_x_n [87](#)
- ssi_clk [86](#)
- ssi_clk_en [87](#)
- SSI_CLK_EN_MODE [77](#)
- SSI_DFLT_FRF [78](#)
- SSI_DFLT_SCPH [78](#)
- SSI_HAS_DMA [77](#)
- SSI_HAS_RX_SAMPLE_DELAY [76](#)
- SSI_HC_FRF [78](#)
- SSI_ID [76](#)
- ssi_in_n [86](#)
- ssi_intr [88](#)
- SSI_INTR_IO [77](#)
- SSI_INTR_POL [77](#)
- SSI_IS_MASTER [75](#)
- ssi_mst_intr [89](#)
- SSI_NUM_SLAVES [76](#)
- ssi_oe_n [86](#)
- ssi_rst_n [86](#)
- SSI_RX_DLY_SR_DEPTH [76](#)
- SSI_RX_FIFO_DEPTH [76](#)
- ssi_rxf_intr [88](#)
- ssi_rxo_intr [89](#)
- ssi_rxu_intr [89](#)
- ssi_sleep [87](#)
- SSI_SYNC_CLK [77](#)
- SSI_TX_FIFO_DEPTH [76](#)
- ssi_txe_intr [88](#)
- ssi_txo_intr [88](#)
- static controller
 - definition [150](#)
- Status register [106](#)
- subsystem
 - definition [150](#)
- synthesis intent
 - definition [150](#)

synthesizable IP

definition [150](#)

T

technology-independent

definition [150](#)

test_DW_apb_ssi.v [125](#)

Testsuite Regression Environment (TRE)

definition [150](#)

Texas Instruments SSP, description of [54](#)

Timing

read operation of DW_apb slave [130](#)

write operation of DW_apb slave [129](#)

Transfer modes

about [34](#)

EEPROM read [35](#)

Receive only [35](#)

Transmit and receive [34](#)

Transmit only [34](#)

Transmit and receive mode [34](#)

Transmit FIFO

buffers [32](#)

interrupts [33](#)

Transmit FIFO level register [105](#)

Transmit FIFO Overflow Interrupt Clear Register [111](#)

Transmit FIFO threshold level register [103](#)

Transmit only mode [34](#)

TRE

definition [150](#)

txd [86](#)

TXFLR [105](#)

TXFTLR [103](#)

TXOICR [111](#)

V

Vera, overview of tests [123](#)

Verification

and Vera tests [123](#)

of DW_apb_ssi coreKit [125](#)

VIP

definition [150](#)

W

workspace

definition [150](#)

wrap

definition [150](#)

wrapper

definition [150](#)

Write coherency

about [132](#)

and asynchronous clocks [136](#)

and identical clocks [134](#)

and synchronous clocks [135](#)

Z

zero-cycle command

definition [150](#)

