

Arria V Hard Processor System Technical Reference Manual

Last updated for Quartus Prime Design Suite: 20.1

av_5v4
2021.05.07



Subscribe



Send Feedback

101 Innovation Drive
San Jose, CA 95134
www.altera.com

ALTERA
now part of Intel

Contents

Arria® V Hard Processor System Technical Reference Manual Revision

History.....	1-1
Introduction to the Hard Processor System.....	2-1
Features of the HPS.....	2-3
HPS Block Diagram and System Integration.....	2-4
HPS Block Diagram.....	2-4
Cortex-A9 MPCore.....	2-5
HPS Interfaces.....	2-5
System Interconnect.....	2-6
On-Chip Memory.....	2-7
Flash Memory Controllers.....	2-8
Support Peripherals.....	2-9
Interface Peripherals.....	2-11
CoreSight Debug and Trace.....	2-14
Endian Support.....	2-14
Introduction to the Hard Processor System Address Map.....	2-15
HPS Address Spaces.....	2-15
HPS Peripheral Region Address Map.....	2-17
Clock Manager.....	3-1
Features of the Clock Manager.....	3-1
Clock Manager Block Diagram and System Integration.....	3-2
L4 Peripheral Clocks.....	3-3
Functional Description of the Clock Manager.....	3-5
Clock Manager Building Blocks.....	3-5
Hardware-Managed and Software-Managed Clocks.....	3-7
Clock Groups.....	3-7
Resets.....	3-17
Safe Mode.....	3-17
Interrupts.....	3-18
Clock Usage By Module.....	3-18
Clock Manager Address Map and Register Definitions.....	3-23
Reset Manager.....	4-1
Reset Manager Block Diagram and System Integration.....	4-2
HPS External Reset Sources.....	4-3
Reset Controller.....	4-4
Module Reset Signals.....	4-5
Slave Interface and Status Register.....	4-10

Functional Description of the Reset Manager.....	4-10
Reset Sequencing.....	4-11
Reset Pins.....	4-15
Reset Effects.....	4-15
Altering Warm Reset System Response.....	4-15
Reset Handshaking.....	4-16
Reset Manager Address Map and Register Definitions.....	4-16
FPGA Manager.....	5-1
Features of the FPGA Manager.....	5-1
FPGA Manager Block Diagram and System Integration.....	5-2
Functional Description of the FPGA Manager.....	5-3
FPGA Manager Building Blocks.....	5-3
FPGA Configuration.....	5-4
FPGA Status.....	5-8
Error Message Extraction.....	5-8
Boot Handshake.....	5-8
General Purpose I/O.....	5-9
Clock.....	5-9
Reset.....	5-9
FPGA Manager Address Map and Register Definitions.....	5-9
System Manager.....	6-1
Features of the System Manager.....	6-1
System Manager Block Diagram and System Integration.....	6-2
Functional Description of the System Manager.....	6-3
Boot Configuration and System Information.....	6-3
Additional Module Control.....	6-3
Boot ROM Code.....	6-6
FPGA Interface Enables.....	6-7
ECC and Parity Control.....	6-8
Preloader Handoff Information.....	6-8
Clocks.....	6-8
Resets.....	6-8
System Manager Address Map and Register Definitions.....	6-9
Scan Manager.....	7-1
Features of the Scan Manager.....	7-1
Scan Manager Block Diagram and System Integration.....	7-2
Arm JTAG-AP Signal Use in the Scan Manager.....	7-2
Arm JTAG-AP Scan Chains.....	7-3
Functional Description of the Scan Manager.....	7-5
Configuring HPS I/O Scan Chains.....	7-5
Communicating with the JTAG TAP Controller.....	7-6
JTAG-AP FIFO Buffer Access and Byte Command Protocol.....	7-6
Clocks.....	7-7

Resets.....	7-8
Scan Manager Address Map and Register Definitions.....	7-8
JTAG-AP Register Name Cross Reference Table.....	7-8
System Interconnect.....	8-1
Features of the System Interconnect.....	8-1
System Interconnect Block Diagram and System Integration.....	8-2
Interconnect Block Diagram.....	8-2
System Interconnect Architecture.....	8-2
Main Connectivity Matrix.....	8-3
Functional Description of the Interconnect.....	8-4
Master to Slave Connectivity Matrix.....	8-4
System Interconnect Address Spaces.....	8-5
Master Caching and Buffering Overrides.....	8-13
Security.....	8-13
Configuring the Quality of Service Logic.....	8-14
Cyclic Dependency Avoidance Schemes.....	8-14
System Interconnect Master Properties.....	8-15
Interconnect Slave Properties.....	8-17
Upsizing Data Width Function.....	8-19
Downsizing Data Width Function.....	8-20
Lock Support.....	8-21
FIFO Buffers and Clock Crossing.....	8-21
System Interconnect Resets.....	8-22
System Interconnect Address Map and Register Definitions.....	8-22
HPS-FPGA Bridges.....	9-1
Features of the HPS-FPGA Bridges.....	9-1
HPS-FPGA Bridges Block Diagram and System Integration.....	9-3
Functional Description of the HPS-FPGA Bridges.....	9-4
The Global Programmers View.....	9-4
Functional Description of the FPGA-to-HPS Bridge.....	9-4
Functional Description of the HPS-to-FPGA Bridge.....	9-7
Functional Description of the Lightweight HPS-to-FPGA Bridge.....	9-10
Clocks and Resets.....	9-14
Data Width Sizing.....	9-15
HPS-FPGA Bridges Address Map and Register Definitions.....	9-16
CoreSight Debug and Trace.....	10-1
Features of CoreSight Debug and Trace.....	10-2
Arm CoreSight Documentation.....	10-2
CoreSight Debug and Trace Block Diagram and System Integration.....	10-3
Functional Description of CoreSight Debug and Trace.....	10-4
Debug Access Port.....	10-4
System Trace Macrocell.....	10-4
Trace Funnel.....	10-5

CoreSight Trace Memory Controller.....	10-5
AMBA Trace Bus Replicator.....	10-6
Trace Port Interface Unit.....	10-6
Embedded Cross Trigger System.....	10-6
Program Trace Macrocell.....	10-11
HPS Debug APB Interface.....	10-11
FPGA Interface.....	10-11
Debug Clocks.....	10-14
Debug Resets.....	10-15
CoreSight Debug and Trace Programming Model.....	10-16
Coresight Component Address.....	10-17
STM Channels.....	10-18
CTI Trigger Connections to Outside the Debug System.....	10-19
Configuring Embedded Cross-Trigger Connections.....	10-21
CoreSight Debug and Trace Address Map and Register Definitions.....	10-22

SDRAM Controller Subsystem.....11-1

Features of the SDRAM Controller Subsystem.....	11-1
SDRAM Controller Subsystem Block Diagram.....	11-2
SDRAM Controller Memory Options.....	11-3
SDRAM Controller Subsystem Interfaces.....	11-4
MPU Subsystem Interface.....	11-4
L3 Interconnect Interface.....	11-4
CSR Interface.....	11-5
FPGA-to-HPS SDRAM Interface.....	11-5
Memory Controller Architecture.....	11-6
Multi-Port Front End.....	11-7
Single-Port Controller.....	11-8
Functional Description of the SDRAM Controller Subsystem.....	11-10
MPFE Operation Ordering.....	11-10
MPFE Multi-Port Arbitration.....	11-10
MPFE SDRAM Burst Scheduling.....	11-13
Single-Port Controller Operation.....	11-14
SDRAM Power Management.....	11-24
DDR PHY.....	11-25
DDR Calibration.....	11-25
Clocks.....	11-25
Resets.....	11-26
Taking the SDRAM Controller Subsystem Out of Reset	11-26
Port Mappings.....	11-26
Initialization.....	11-27
FPGA-to-SDRAM Protocol Details.....	11-28
SDRAM Controller Subsystem Programming Model.....	11-32
HPS Memory Interface Architecture.....	11-32
HPS Memory Interface Configuration.....	11-32
HPS Memory Interface Simulation.....	11-33
Generating a Preloader Image for HPS with EMIF.....	11-34
Debugging HPS SDRAM in the Preloader.....	11-35

Enabling UART or Semihosting Printout.....	11-35
Enabling Simple Memory Test.....	11-36
Enabling the Debug Report.....	11-37
Writing a Predefined Data Pattern to SDRAM in the Preloader.....	11-40
SDRAM Controller Address Map and Register Definitions.....	11-41
On-Chip Memory.....	12-1
On-Chip RAM.....	12-1
Features of the On-Chip RAM.....	12-1
On-Chip RAM Block Diagram and System Integration.....	12-1
Functional Description of the On-Chip RAM.....	12-2
Boot ROM.....	12-3
Features of the Boot ROM.....	12-3
Boot ROM Block Diagram and System Integration.....	12-3
Functional Description of the Boot ROM.....	12-3
On-Chip Memory Address Map and Register Definitions.....	12-4
NAND Flash Controller.....	13-1
NAND Flash Controller Features.....	13-1
NAND Flash Controller Block Diagram and System Integration.....	13-2
NAND Flash Controller Signal Descriptions.....	13-2
Functional Description of the NAND Flash Controller.....	13-3
Discovery and Initialization.....	13-3
Bootstrap Interface.....	13-4
Configuration by Host.....	13-5
Local Memory Buffer.....	13-6
Clocks.....	13-6
Resets.....	13-7
Indexed Addressing.....	13-7
Command Mapping.....	13-8
Data DMA.....	13-14
ECC.....	13-18
NAND Flash Controller Programming Model.....	13-21
Basic Flash Programming.....	13-22
Flash-Related Special Function Operations.....	13-25
NAND Flash Controller Address Map and Register Definitions.....	13-33
SD/MMC Controller.....	14-1
Features of the SD/MMC Controller.....	14-1
SD Card Support Matrix.....	14-2
MMC Support Matrix.....	14-3
SD/MMC Controller Block Diagram and System Integration.....	14-3
SD/MMC Controller Signal Description.....	14-4
Functional Description of the SD/MMC Controller.....	14-5
SD/MMC/CE-ATA Protocol.....	14-5
BIU.....	14-6

CIU.....	14-20
Clocks.....	14-35
Resets.....	14-36
Voltage Switching.....	14-37
SD/MMC Controller Programming Model.....	14-39
Software and Hardware Restrictions [†]	14-39
Initialization [†]	14-41
Controller/DMA/FIFO Buffer Reset Usage.....	14-47
Enabling ECC.....	14-47
Enabling FIFO Buffer ECC.....	14-47
Non-Data Transfer Commands.....	14-48
Data Transfer Commands.....	14-50
Transfer Stop and Abort Commands.....	14-56
Internal DMA Controller Operations.....	14-58
Commands for SDIO Card Devices.....	14-60
CE-ATA Data Transfer Commands.....	14-62
Card Read Threshold.....	14-70
Interrupt and Error Handling.....	14-73
Bootling Operation for eMMC and MMC.....	14-74
SD/MMC Controller Address Map and Register Definitions.....	14-84
Quad SPI Flash Controller.....	15-1
Features of the Quad SPI Flash Controller.....	15-1
Quad SPI Flash Controller Block Diagram and System Integration.....	15-2
Interface Signals.....	15-3
Functional Description of the Quad SPI Flash Controller.....	15-3
Overview.....	15-3
Data Slave Interface.....	15-4
SPI Legacy Mode.....	15-7
Register Slave Interface.....	15-8
Local Memory Buffer.....	15-8
DMA Peripheral Request Controller.....	15-9
Arbitration between Direct/Indirect Access Controller and STIG.....	15-10
Configuring the Flash Device.....	15-10
XIP Mode.....	15-12
Write Protection.....	15-12
Data Slave Sequential Access Detection.....	15-13
Clocks.....	15-13
Resets.....	15-13
Interrupts.....	15-14
Quad SPI Flash Controller Programming Model.....	15-15
Setting Up the Quad SPI Flash Controller.....	15-16
Indirect Read Operation with DMA Disabled.....	15-16
Indirect Read Operation with DMA Enabled.....	15-17
Indirect Write Operation with DMA Disabled.....	15-17
Indirect Write Operation with DMA Enabled.....	15-18
XIP Mode Operations.....	15-18
Quad SPI Flash Controller Address Map and Register Definitions.....	15-20

DMA Controller.....	16-1
Features of the DMA Controller.....	16-1
DMA Controller Block Diagram and System Integration.....	16-3
Functional Description of the DMA Controller.....	16-3
Peripheral Request Interface.....	16-4
DMA Controller Address Map and Register Definitions.....	16-7
Address Map and Register Definitions.....	16-9
 Ethernet Media Access Controller.....	 17-1
Features of the Ethernet MAC.....	17-2
MAC.....	17-2
DMA.....	17-2
Management Interface.....	17-3
Acceleration.....	17-3
PHY Interface.....	17-3
EMAC Block Diagram and System Integration.....	17-4
EMAC Signal Description.....	17-5
HPS EMAC I/O Signals.....	17-6
FPGA EMAC I/O Signals.....	17-8
PHY Management Interface.....	17-11
EMAC Internal Interfaces.....	17-12
DMA Master Interface.....	17-12
Timestamp Interface.....	17-13
Functional Description of the EMAC.....	17-14
Transmit and Receive Data FIFO Buffers.....	17-15
DMA Controller.....	17-16
Descriptor Overview.....	17-29
IEEE 1588-2002 Timestamps.....	17-46
IEEE 1588-2008 Advanced Timestamps.....	17-52
IEEE 802.3az Energy Efficient Ethernet.....	17-55
Checksum Offload.....	17-56
Frame Filtering.....	17-56
Clocks and Resets.....	17-61
Interrupts.....	17-63
Ethernet MAC Programming Model.....	17-63
System Level EMAC Configuration Registers.....	17-63
EMAC FPGA Interface Initialization.....	17-65
EMAC HPS Interface Initialization.....	17-66
DMA Initialization.....	17-67
EMAC Initialization and Configuration.....	17-68
Performing Normal Receive and Transmit Operation.....	17-68
Stopping and Starting Transmission.....	17-69
Programming Guidelines for Energy Efficient Ethernet.....	17-69
Programming Guidelines for Flexible Pulse-Per-Second (PPS) Output.....	17-70
Ethernet MAC Address Map and Register Definitions.....	17-72

USB 2.0 OTG Controller.....	18-1
Features of the USB OTG Controller.....	18-2
Supported PHYS.....	18-3
USB OTG Controller Block Diagram and System Integration.....	18-4
USB 2.0 ULPI PHY Signal Description.....	18-5
Functional Description of the USB OTG Controller.....	18-6
USB OTG Controller Block Description.....	18-6
Local Memory Buffer.....	18-10
Clocks.....	18-10
Resets.....	18-10
Interrupts.....	18-11
USB OTG Controller Programming Model.....	18-13
Enabling ECC.....	18-13
Enabling SPRAM ECCs.....	18-13
Host Operation.....	18-14
Device Operation.....	18-15
USB 2.0 OTG Controller Address Map and Register Definitions.....	18-17
USB Data FIFO Address Map.....	18-17
USB Direct Access FIFO RAM Address Map.....	18-19
 SPI Controller.....	 19-1
Features of the SPI Controller.....	19-1
SPI Block Diagram and System Integration.....	19-2
SPI Block Diagram.....	19-2
SPI Controller Signal Description.....	19-3
Interface to HPS I/O.....	19-3
FPGA Routing.....	19-3
Functional Description of the SPI Controller.....	19-4
Protocol Details and Standards Compliance.....	19-4
SPI Controller Overview.....	19-5
Transfer Modes.....	19-8
SPI Master.....	19-9
SPI Slave.....	19-12
Partner Connection Interfaces.....	19-16
DMA Controller Interface.....	19-21
Slave Interface.....	19-21
Clocks and Resets.....	19-22
SPI Programming Model.....	19-23
Master SPI and SSP Serial Transfers.....	19-24
Master Microwire Serial Transfers.....	19-26
Slave SPI and SSP Serial Transfers.....	19-28
Slave Microwire Serial Transfers.....	19-29
Software Control for Slave Selection.....	19-29
DMA Controller Operation.....	19-30
SPI Controller Address Map and Register Definitions.....	19-33

I²C Controller.....	20-1
Features of the I ² C Controller.....	20-1
I ² C Controller Block Diagram and System Integration.....	20-2
I ² C Controller Signal Description.....	20-3
Functional Description of the I ² C Controller.....	20-4
Feature Usage.....	20-4
Behavior.....	20-5
Protocol Details.....	20-6
Multiple Master Arbitration.....	20-11
Clock Frequency Configuration.....	20-13
SDA Hold Time.....	20-15
DMA Controller Interface.....	20-15
Clocks.....	20-15
Resets.....	20-15
I ² C Controller Programming Model.....	20-16
Slave Mode Operation.....	20-16
Master Mode Operation.....	20-19
Disabling the I ² C Controller.....	20-20
DMA Controller Operation.....	20-20
I ² C Controller Address Map and Register Definitions.....	20-24
 UART Controller.....	 21-1
UART Controller Features.....	21-1
UART Controller Block Diagram and System Integration.....	21-2
UART Controller Signal Description.....	21-3
HPS I/O Pins.....	21-3
FPGA Routing.....	21-3
Functional Description of the UART Controller.....	21-4
FIFO Buffer Support.....	21-4
UART(RS232) Serial Protocol.....	21-5
Automatic Flow Control.....	21-5
Clocks.....	21-7
Resets.....	21-7
Interrupts.....	21-7
DMA Controller Operation.....	21-10
Transmit FIFO Underflow.....	21-11
Transmit Watermark Level.....	21-11
Transmit FIFO Overflow.....	21-12
Receive FIFO Overflow.....	21-13
Receive Watermark Level.....	21-13
Receive FIFO Underflow.....	21-13
UART Controller Address Map and Register Definitions.....	21-14
 General-Purpose I/O Interface.....	 22-1
Features of the GPIO Interface.....	22-1

GPIO Interface Block Diagram and System Integration.....	22-2
Functional Description of the GPIO Interface.....	22-2
Debounce Operation.....	22-2
Pin Directions.....	22-3
Taking the GPIO Interface Out of Reset	22-3
GPIO Pin State During Reset.....	22-4
GPIO Interface Programming Model.....	22-4
General-Purpose I/O Interface Address Map and Register Definitions.....	22-4
Timer	23-1
Features of the Timer.....	23-1
Timer Block Diagram and System Integration.....	23-1
Functional Description of the Timer.....	23-2
Clocks.....	23-3
Resets.....	23-3
Interrupts.....	23-3
FPGA Interface.....	23-3
Timer Programming Model.....	23-5
Initialization.....	23-5
Enabling the Timer.....	23-5
Disabling the Timer.....	23-5
Loading the Timer Countdown Value.....	23-5
Servicing Interrupts.....	23-6
Timer Address Map and Register Definitions.....	23-6
Watchdog Timer.....	24-1
Features of the Watchdog Timer.....	24-1
Watchdog Timer Block Diagram and System Integration.....	24-2
Functional Description of the Watchdog Timer.....	24-2
Watchdog Timer Counter.....	24-2
Watchdog Timer Pause Mode.....	24-3
Watchdog Timer Clocks.....	24-3
Watchdog Timer Resets.....	24-3
FPGA Interface.....	24-4
Watchdog Timer Programming Model.....	24-4
Setting the Timeout Period Values.....	24-4
Selecting the Output Response Mode.....	24-4
Enabling and Initially Starting a Watchdog Timer.....	24-5
Reloading a Watchdog Counter.....	24-5
Pausing a Watchdog Timer.....	24-5
Disabling and Stopping a Watchdog Timer.....	24-5
Watchdog Timer State Machine.....	24-5
Watchdog Timer Address Map and Register Definitions.....	24-7
Introduction to the HPS Component.....	25-1
MPU Subsystem.....	25-2

Arm CoreSight Debug Components.....	25-2
Interconnect.....	25-2
HPS-to-FPGA Interfaces.....	25-2
Memory Controllers.....	25-3
Support Peripherals.....	25-3
Interface Peripherals.....	25-3
On-Chip Memories.....	25-4
Instantiating the HPS Component.....	26-1
FPGA Interfaces.....	26-1
General Interfaces.....	26-2
FPGA-to-HPS SDRAM Interface.....	26-3
DMA Peripheral Request.....	26-5
Interrupts.....	26-5
AXI Bridges.....	26-7
Configuring HPS Clocks and Resets.....	26-8
User Clocks.....	26-8
Reset Interfaces.....	26-9
PLL Reference Clocks.....	26-10
Peripheral FPGA Clocks.....	26-11
Configuring Peripheral Pin Multiplexing.....	26-11
Configuring Peripherals.....	26-12
Connecting Unassigned Pins to GPIO.....	26-12
Using Unassigned IO as LoanIO.....	26-13
Resolving Pin Multiplexing Conflicts.....	26-13
Peripheral Signals Routed to FPGA	26-14
Configuring the External Memory Interface.....	26-14
Selecting PLL Output Frequency and Phase.....	26-15
Using the Address Span Extender Component.....	26-15
Generating and Compiling the HPS Component.....	26-16
HPS Component Interfaces.....	27-1
Memory-Mapped Interfaces.....	27-1
FPGA-to-HPS Bridge.....	27-1
HPS-to-FPGA and Lightweight HPS-to-FPGA Bridges.....	27-2
FPGA-to-HPS SDRAM Interface.....	27-3
Clocks.....	27-4
Alternative Clock Inputs to HPS PLLs.....	27-4
User Clocks.....	27-4
AXI Bridge FPGA Interface Clocks.....	27-5
SDRAM Clocks.....	27-5
Peripheral FPGA Clocks.....	27-5
Resets.....	27-6
HPS-to-FPGA Reset Interfaces.....	27-6
HPS External Reset Request.....	27-6
Peripheral Reset Interfaces.....	27-7
Debug and Trace Interfaces.....	27-7

Trace Port Interface Unit.....	27-7
FPGA System Trace Macrocell Events Interface.....	27-7
FPGA Cross Trigger Interface.....	27-7
Debug APB Interface.....	27-7
Peripheral Signal Interfaces.....	27-7
DMA Controller Interface.....	27-7
Other Interfaces.....	27-8
MPU Standby and Event Interfaces.....	27-8
General Purpose Signals.....	27-9
FPGA-to-HPS Interrupts.....	27-9
Boot from FPGA Interface.....	27-9
Input-only General Purpose Interface.....	27-9
Simulating the HPS Component.....	28-1
Simulation Flows.....	28-2
Setting Up the HPS Component for Simulation.....	28-3
Generating the HPS Simulation Model in Platform Designer (Standard).....	28-5
Running the Simulations.....	28-5
Clock and Reset Interfaces.....	28-9
Clock Interface.....	28-9
Reset Interface.....	28-10
FPGA-to-HPS AXI Slave Interface.....	28-11
HPS-to-FPGA AXI Master Interface.....	28-12
Lightweight HPS-to-FPGA AXI Master Interface.....	28-12
FPGA-to-HPS SDRAM Interface.....	28-13
HPS Memory Interface Simulation.....	28-13
HPS-to-FPGA MPU Event Interface.....	28-14
Interrupts Interface.....	28-14
HPS-to-FPGA Debug APB Interface.....	28-16
FPGA-to-HPS System Trace Macrocell Hardware Event Interface.....	28-16
HPS-to-FPGA Cross-Trigger Interface.....	28-17
HPS-to-FPGA Trace Port Interface.....	28-17
FPGA-to-HPS DMA Handshake Interface.....	28-18
Boot from FPGA Interface.....	28-19
General Purpose Input Interface.....	28-20
Booting and Configuration.....	A-1
Boot Overview.....	A-1
FPGA Configuration Overview.....	A-1
Booting and Configuration Options.....	A-2
Boot Definitions.....	A-5
Reset.....	A-5
Boot ROM.....	A-5
Boot Select.....	A-6
Flash Memory Devices for Booting.....	A-13
Clock Select.....	A-25
I/O Configuration.....	A-25

L4 Watchdog 0 Timer.....	A-26
Preloader.....	A-26
U-Boot Loader.....	A-26
Boot ROM Flow.....	A-26
Typical Preloader Boot Flow.....	A-28
HPS State on Entry to the Preloader.....	A-31
Shared Memory.....	A-31
Loading the Preloader Image.....	A-32
FPGA Configuration.....	A-33
Full Configuration.....	A-33
Partial Reconfiguration.....	A-34

Arria® V Hard Processor System Technical Reference Manual Revision History

1

2021.05.07

av_5v4



Subscribe



Send Feedback

Table 1-1: Arria® V Hard Processor System Technical Reference Manual Revision History Summary

Chapter	Date of Last Update
Introduction to the Hard Processor System	October 28, 2016
Clock Manager	November 2, 2015
Reset Manager	November 2, 2015
FPGA Manager	June 14, 2019
System Manager	July 17, 2018
Scan Manager	May 3, 2016
System Interconnect	May 4, 2015
HPS-FPGA Bridges	September 3, 2020
Cortex®-A9 Microprocessor Unit Subsystem	September 3, 2020
CoreSight* Debug and Trace	July 31, 2014
SDRAM Controller Subsystem Controller	February 28, 2020
On-Chip Memory	January 26, 2018
NAND Flash Controller	January 26, 2018
SD/MMC Controller	May 7, 2021
Quad SPI Flash Controller	September 3, 2020
DMA Controller	January 26, 2018
Ethernet Media Access Controller	April 9, 2021
USB 2.0 OTG Controller	January 26, 2018
SPI Controller	January 26, 2018
I²C Controller	May 4, 2015
UART Controller	November 2, 2015

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

Chapter	Date of Last Update
General-Purpose I/O Interface	September 3, 2020
Timer	June 30, 2014
Watchdog Timer	November 2, 2015
Introduction to the HPS Component	December 30, 2013
Instantiating the HPS Component	November 2, 2015
HPS Component Interfaces	May 4, 2015
Simulating the HPS Component	May 3, 2016
Bootting and Configuration	September 3, 2020

Document Version	Changes
2016.10.28	- Added 8-bit support for eMMC for SD/MMC - Renamed MPU Subsystem to Cortex-A9 MPCore*
2016.05.03	Maintenance release.
2015.11.02	Updated the link to the Memory Maps.
2015.05.04	Corrected the base address for NANDDATA in the "Peripheral Region Address Map" table.
2014.12.15	Maintenance release
2014.07.31	Updated address maps and register descriptions
2014.06.30	Maintenance release
2014.02.28	Maintenance release
2013.12.30	Maintenance release
1.3	Minor updates.
1.2	Updated address spaces section.
1.1	Added peripheral region address map.
1.0	Initial release.

[Introduction to the Hard Processor System](#) on page 2-1

Document Version	Changes
2020.01.13	Correct typical sdmmc_clk frequencies in <i>Flash Controller Clocks</i>
2015.11.02	Minor formatting updates.

Document Version	Changes
2015.05.04	Minor formatting updates.
2014.12.15	FREF, FVCO, and FOUT Equations section updated. More information added about vco register, M and N equations. Reference Clock information added to Clock Groups section.
2014.06.30	E0SC1 changed to HPS_CLK1 E0SC2 changed to HPS_CLK2 Added Address Map and Register Descriptions
2014.02.28	Updated content in the "Peripheral Clock Group" section
2013.12.30	Minor formatting updates.
1.2	Minor updates.
1.1	• Reorganized and expanded functional description section. • Added address map and register definitions section.
1.0	Initial release.

[Clock Manager](#) on page 3-1

Document Version	Changes
2015.11.02	Updated "Reset Pins" section
2015.05.04	Updated: • MISC Group, Generated Module Resets table • "Reset Pins" section
2014.12.15	• Signal power information added to "HPS External Reset Sources" section • Updated block diagram with h2f_dbg_rst_n signal
2014.06.30	• Updated "Functional Description of Reset Manager" • Added address map and register descriptions
2014.02.28	Updated sections: • Reset Sequencing • Warm Reset Assertion Sequence
2013.12.30	Minor formatting issues.

Document Version	Changes
1.2	Added cold and warm reset timing diagrams.
1.1	Added reset controller, functional description, and address map and register definitions sections.
1.0	Initial release.

Reset Manager on page 4-1

Document Version	Changes
2019.06.14	Corrected the msel descriptions for encodings 0x0 through 0x2 and 0x4 to 0x6 in the stat register.
2015.11.02	Provided more information for the configuration schemes for the dedicated pins.
2015.05.04	Added information about FPPx32.
2014.12.15	Maintenance release
2014.06.30	Added address maps and register definitions
2014.02.28	Maintenance release
2013.12.30	Minor updates.
1.3	Minor updates.
1.2	Updated the FPGA configuration section.
1.1	- Updated the configuration schemes table. - Updated the FPGA configuration section. - Added address map and register definitions section.
1.0	Initial release.

FPGA Manager on page 5-1

Document Version	Changes
2018.11.03	Modified mode register bitfield descriptions for clarity.
2018.07.17	Made the following changes to the Pin Mux Control Group register block: Added new registers in the Pin Mux Control Group for routing QSPI, SD/MMC, UART, I2C, and SPI signals to the FPGA.

Document Version	Changes
2014.06.30	- Added address map and register descriptions - Updated ECC Parity Control
2014.02.28	Maintenance release
2013.12.30	Maintenance release.
1.2	Minor updates.
1.1	Added functional description, address map and register definitions sections.
1.0	Initial release.

[System Manager](#) on page 6-1

Document Version	Changes
2016.05.03	Added a list of the HPS I/O pins that do not support boundary scan tests in the <i>Arm* JTAG-AP Scan Chains</i> section.
2015.11.02	Maintenance release
2015.05.04	Maintenance release
2014.12.15	Maintenance release
2014.06.30	- Add address map and register definitions - Remove erroneous reference to CAN controller
2014.02.28	Update to "Scan Manager Block Diagram and System Integration" section
2013.12.30	Minor formatting issues
1.2	Added JTAG-AP descriptions.
1.1	Added block diagram and system integration, functional description, and address map and register definitions sections.
1.0	Initial release.

[Scan Manager](#) on page 7-1

Document Version	Changes
2015.05.04	- Reference AXI ID encoding in MPU chapter - Add information about the SDRAM address space
2014.12.15	- Minor correction to table in "Available Address Maps" - Add detail to "L3 Address Space"
2014.06.30	- Corrected master interconnect security properties for: - Ethernet MAC - ETR - Added address map and register descriptions
2014.02.28	Maintenance release
2013.12.30	Maintenance release
1.2	Minor updates.
1.1	- Added interconnect connectivity matrix. - Rearranged functional description sections. - Simplified address remapping section. - Added address map and register definitions section.
1.0	Initial release.

System Interconnect on page 8-1

Document Version	Changes
2020.09.03	Updated <i>Taking HPS-FPGA Bridges Out of Reset</i> with clarification on the state of the HPS GPIO during cold reset.
2014.06.30	Added address maps and register definitions
2014.02.28	Maintenance release
2013.12.30	Maintenance release
1.1	Described GPV
1.0	Initial release

HPS-FPGA Bridges on page 9-1

Document Version	Changes
2020.09.03	Added Interconnect Master (L2M0) to the "HPS Peripheral Master Input IDs" table in <i>HPS Peripheral Master Input IDs</i> .
2020.01.13	Added new section <i>Avoiding ACP Dependency Lockup</i>
2019.06.14	Added details about arbitration behavior in the SCU when the ACP is not being used in the <i>Implementation Details</i> of the <i>Snoop Control Unit</i> section,
2016.10.28	- Added note to "AXI Master Configuration for ACP Access" section - Added "Configuring AxCACHE[3:0] Sideband Signals" and "Configuring AxUser[4:0] Sideband Signals" subsections to the "AXI Master Configuration for ACP Access" section - Added note in the "Implementation Details" subsection of the "ACP ID Mapper" section.
2016.05.03	Maintenance release
2015.11.02	- Reordered "L2 Cache" subsections - Renamed "ECC Support" L2 subsection to be "Single Event Upset Protection" - Added "L2 Cache Parity" subsection in "L2 Cache" section
2015.05.04	Clarified EMAC0 and EMAC1 ACP Mapper IDs in the "HPS Peripheral Master Input IDs" table in the "HPS Peripheral Master Input IDs" section.
2014.12.15	- Added bus transaction scenarios in the "Accelerator Coherency Port" section - Added the "AxUSER and AxCACHE Attributes" subsection to the "Accelerator Coherency Port" section - Added the "Shared Requests on ACP" subsection to the "Accelerator Coherency Port" section - Added the "Configuration for ACP Use" subsection to the "Accelerator Coherency Port" section - Clarified how to use fixed mapping mode in the ACP ID Mapper - Updated HPS Peripheral Master Input IDs table - Added a note to the "Control of the AXI User Sideband Signals" subsection in the "ACP ID Mapper" section. - Added parity error handling information to the "L1 Caches" section and the "Cache Controller Configuration" topic of the "L2 Cache" section.
2014.06.30	- Added Reset Section to Cortex-A9 Processor - Updated HPS Peripheral Master Input IDs table - Added ACP ID Mapper Address Map and Register Definitions - Added information in ECC Support section regarding ECC errors - Minor clarifications regarding MPU description and module revision numbers
2014.02.28	Maintenance release
2013.12.30	Correct SDRAM region address in Arm Cortex-A9 MPCoreAddress Map
1.2	Minor updates.

Document Version	Changes
1.1	- Add description of the ACP ID mapper - Consolidate redundant information
1.0	Initial release.

Cortex-A9 Microprocessor Unit Subsystem

Document Version	Changes
2014.07.31	Updated the address map and register definitions.
2014.06.30	Added address map and register definitions.
2014.02.28	Maintenance release.
2013.12.30	Maintenance release.
1.2	Minor updates.
1.1	Added functional description, programming model, and address map and register definition sections.
1.0	Initial release.

CoreSight Debug and Trace on page 10-1

Document Version	Changes
2020.02.28	In the <i>Memory Protection</i> section - Corrected the "Protection" field definition in the "Fields for Rules in Memory Protection Table".
2018.07.17	Modified text to clarify that there is support for up to 4 Gb external memory device per chip select.

Document Version	Changes
2015.11.02	- Added information regarding calculation of ECC error byte address location from erraddr register in "User Notification of ECC Errors" section - Added information regarding bus response to memory protection transaction failure in "Memory Protection" section - Clarified "Protection" row in "Fields for Rules in Memory Protection" table in the "Memory Protection" section - Clarified protruledata.security column in "Rules in Memory Protection Table for Example Configuration" table in the "Example of Configuration for TrustZone" section - Added note about double-bit error functionality in "ECC Write Backs" subsection of "ECC" section - Added the "DDR Calibration" subsection under "DDR PHY" section
2015.05.04	Added the recommended sequence for writing or reading a rule in the "Memory Protection" section.
2014.12.15	- Added SDRAM Protection Access Flow Diagram to "Memory Protection" subsection in the "Single-Port Controller Operation" section. - Changed the "SDRAM Multi-Port Scheduling" section to "SDRAM Multi-Port Arbitration" and added detailed information on how to use and program the priority and weighted arbitration scheme.
2014.6.30	- Added <i>Port Mappings</i> section. - Added <i>SDRAM Controller Memory Options</i> section. - Enhanced <i>Example of Configuration for TrustZone</i> section. - Added SDRAM Controller address map and registers.
2013.12.30	- Added <i>Generating a Preloader Image for HPS with EMIF</i> section. - Added <i>Debugging HPS SDRAM in the Preloader</i> section. - Enhanced <i>Simulation</i> section.
1.1	Added address map and register definitions section.
1.0	Initial release.

SDRAM Controller Subsystem on page 11-1

Document Version	Changes
2018.01.26	Updated "On-Chip RAM Initialization" section with steps to enable ECC.
2016.10.28	Maintenance release
2016.05.03	Maintenance release
2015.11.02	Maintenance release
2015.05.04	Maintenance release

Document Version	Changes
2014.12.15	Maintenance release
2014.06.30	Added address maps and register definitions
2014.02.28	Maintenance release
2013.12.30	Maintenance release
1.1	Added address map section
1.0	Initial release

On-Chip Memory on page 12-1

Document Version	Changes
2018.01.26	Updated "ECC Enabling" section with steps to enable ECC.
2016.10.28	Added content about the local memory buffer
2016.05.27	Added a link to the <i>Supported Flash Devices for Cyclone V and Arria V SoC</i> webpage.
2016.05.03	Maintenance release
2015.11.02	- Moved "Interface Signals" section after "NAND Flash Controller Block Diagram and System Integration" section and renamed it to "NAND Flash Controller Signal Description" - Updated the Interrupt and DMA Enabling section to recommend reading back a register to ensure clearing an interrupt status - Specified the valid values for Burst Length in the Command-Data Pair 4 table - Updated the description of dma_cmd_comp and added a RESERVED bit for intr_status0/1/2/3 and intr_en0/1/2/3
2015.05.04	Added information about clearing out the ECC before the feature is enabled
2014.12.15	Maintenance release
2014.07.31	Updated address map and register definitions.
2014.06.30	- Added address map and register definitions. - Removed Command DMA section.
2014.02.28	Maintenance release
2013.12.30	Maintenance release
1.2	- Supports one 8-bit device - Show additional supported block sizes - Bad block marker handling

Document Version	Changes
1.1	Added programming model section.
1.0	Initial release

NAND Flash Controller on page 13-1

Document Version	Changes
2021.05.07	Corrected the Max Data Rate for MMCPlus and eMMC.
2018.01.26	Added "Enabling ECC" section.
2017.12.27	Added 8-bit support for eMMC in the "Features of SD/MMC Controller" section. (FB320076)
2016.10.28	- Removed SPI support in tables in the Features section - Added 8-bit support for eMMC for SD/MMC
2016.05.27	Added a link to the <i>Supported Flash Devices for Cyclone V and Arria V SoC</i> webpage.
2016.05.03	Maintenance release.
2015.11.02	- Moved "Interface Signals" section below "SD/MMC Controller Block Diagram and System Integration" section and renamed to "SD/MMC Signal Description." Clarified signals in this section. - Added information that Card Detect is only supported on interfaces routed via the FPGA fabric.
2015.05.04	Added information about clearing out the ECC before the feature is enabled
2014.12.15	Maintenance release
2014.06.30	Added address maps and register definitions
2014.02.28	Maintenance release
2013.12.30	Maintenance release
1.1	- Added programming model section. - Reorganized programming information. - Added information about ECCs. - Added pin listing. - Updated clocks section.
1.0	Initial release.

SD/MMC Controller on page 14-1

Document Version	Changes
2020.09.03	Updated the definition for the QSPI register: indaddrtrig in the <i>Quad SPI Flash Controller Address Map and Register Definitions</i> section
2019.07.09	Added a new section, <i>Write Request</i> , with WREN and RDSR information
2019.06.14	Maintenance release
2018.01.26	Updated "Local Memory Buffer" section with steps to enable ECC.
2016.10.28	Maintenance release
2016.05.27	- Changed the name of the internal QSPI reference clock from <code>qspi_clk</code> to <code>qspi_ref_clk</code>; and the external QSPI output clock, from <code>sc_clk_out</code> to <code>qspi_clk</code>. - Added a link to the <i>Supported Flash Devices for Cyclone V and Arria V SoC</i> webpage. - Re-worded information about disabling the watermark feature in the "Indirect Read Operation" and "Indirect Write Operation" sections.
2016.05.03	- Added clarification for the SRAM indirect read and write size allocations. - Updated the SRAM block on the "Quad SPI Flash Controller Block Diagram and System Integration" figure.
2015.11.02	- Moved "Interface Signals" section below "Quad SPI Flash Controller Block Diagram and System Integration" - Better defined <code>l4_mp_clk</code> clock.
2015.05.04	Added information about clearing out the ECC before the feature is enabled
2014.12.15	Maintenance release
2014.07.31	Updated address maps and register descriptions
2014.06.30	Added address maps and register definitions
2014.02.28	Maintenance release
2013.12.30	Maintenance release
1.2	Minor updates.
1.1	Added block diagram and system integration, functional description, programming model, and address map and register definitions sections.
1.0	Initial release.

[Quad SPI Flash Controller](#) on page 15-1

Document Version	Changes
2018.01.26	Updated "Initializing and Clearing of Memory before Enabling ECC" section with steps to enable ECC.

Document Version	Changes
2016.10.28	Maintenance release
2016.05.03	Maintenance release
2015.11.02	- Updated link point to the HPS Address Map and Register Definitions - Added information about the instruction fetch cache properties - Added a description about the relationship between the GIC interrupt map and INTCLR register
2015.05.04	Added Synopsys* handshake rules.
2014.12.15	Maintenance release
2014.07.31	Updated address maps and register descriptions
2014.06.30	Added address maps and register definitions
2014.02.28	ECC updates
1.2	Maintenance release
1.1	Minor updates
1.0	Initial release

[DMA Controller](#) on page 16-1

Document Version	Changes
2021.04.09	Added <code>emac_clk_tx_i</code> handling requirement for exported HPS EMAC GMII interface in the <i>EMAC FPGA Interface Initialization</i> section.
2020.08.18	Updated <i>EMAC HPS Interface Initialization</i> to clarify how to verify RX PHY clocks after bringing the Ethernet PHY out of reset.
2019.06.14	Clarified the PCF bit description for encoding value 0x2 in the MAC_Frame_Filter register. - Clarified "Busy Bit" (gb bit of GMII_Address register) in Flow_Control register description. - Clarified that ttc bit resides in the Operation_Mode Register (Register 6). - Clarified that the pcsancis and the pcslchgis bits of the Interrupt_Status register can be ignored because they apply to TBI, RTBI, or SGMII interface only.
2016.10.28	- Note added into <i>PHY Interface</i> section - Bit 16 updated Transmit Descriptor table
2016.05.03	Maintenance release.

Document Version	Changes
2015.11.02	- Added the following subsections in the "Layer 3 and Layer 4 Filters" section: - Layer 3 and Layer 4 Filters Register Set - Layer 3 Filtering - Layer 4 Filtering
2015.05.04	- Corrected IEEE 1588 timestamp resolution in the "EMAC Block Diagram and System Integration" section and the "IEEE 1588-2002 Timestamps" section - Added reset pulse width for <code>rst_clk_tx_n_o</code> and <code>rst_clk_rx_n_o</code> in the "FPGA EMAC I/O signals" section - Added subsections "Ordinary Clock," "Boundary Clock," "End-to-End Transparent Clock" and "Peer-to-Peer Transparent Clock" in the "Clock Type" section
2014.12.15	- Updated <i>EMAC Block Diagram and System Integration</i> section with new diagram and information. - Added <i>Signal Descriptions</i> section. - Added <i>EMAC Internal Interfaces</i> section. - Added <i>TX FIFO</i> and <i>RX FIFO</i> subsection to the <i>Transmit and Receive Data FIFO Buffers</i> section. - Updated <i>Descriptor Overview</i> section to clarify support for only enhanced (alternate) descriptors. - Added <i>Destination and Source Address Filtering Summary</i> in <i>Frame Filtering</i> Section. - Added <i>Clock Structure</i> sub-section to <i>Clocks and Resets</i> section - Added <i>System Level EMAC Configuration Registers</i> section in <i>Ethernet Programming Model</i> - Added <i>EMAC Interface Initialization for FPGA GMII/MII Mode</i> section in Ethernet Programming Model - Added <i>EMAC Interface Initialization for RGMII/RMII Mode</i> section in Ethernet Programming Model - Corrected <i>DMA Initialization</i> and <i>EMAC Initialization and Configuration</i> titles to appear on correct initialization information - Removed duplicate programming information for DMA - Added <i>Taking the Ethernet MAC Out of Reset</i> section.
2014.06.30	Updated EMAC to RGMII Interface table with EMAC Port names Updated EMAC to FPGA PHY Interface table with Signal names Updated EMAC to FPGA IEEE1588 Timestamp Interface with Signal names Added Address Map and Register Descriptions
2014.02.28	ECC updates.
1.4	Maintenance release.
1.3	- Expanded shared memory block table. - Added CSEL tables. - Additional minor updates.
1.2	Updated the HPS boot and FPGA configuration sections.

Ethernet Media Access Controller on page 17-1

Document Version	Changes
2018.01.26	Added steps for enabling ECC.
2016.10.28	Maintenance release.
2016.05.03	Maintenance release.
2015.11.02	Renamed "ULPI PHY Interface" section to "USB 2.0 ULPI PHY Signal Description" and moved it after the "USB OTG Controller Block Diagram and System Integration" section.
2015.05.04	Maintenance release.
2014.12.15	• Maintenance release. • Added <i>Taking the USB OTG Out of Reset</i> section.
2014.07.31	Updated address map and register definitions.
2014.06.30	Added USB OTG Controller address map and register definitions.
2014.02.28	Maintenance release.
2013.12.30	Maintenance release.
1.2	• Described interrupt generation. • Described software initialization in host and device modes. • Described software operation in host and device modes. • Simplified features list. • Simplified hardware description.
1.1	Added information about ECCs.
1.0	Initial release.

USB 2.0 OTG Controller on page 18-1

Document Version	Changes
2017.01.26	Corrected the support information for continuous data transfers in <i>SPI Serial Format</i> .
2016.10.28	Maintenance release.
2016.05.03	Maintenance release.

Document Version	Changes
2015.11.02	- Renamed "Interface Pins" section to "Interface to HPS I/O" and moved it under the "SPI Controller Signal Description" section - Moved "FPGA Routing" section under "SPI Controller Signal Description" section - Added Multi-Master mode to "Features of the SPI Controller" section - Updated "RXD Sample Delay" section - Updated "SPI Slave" section - Updated "Glue Logic for Master Port ss_in_n" section
2015.05.04	Maintenance release.
2014.12.15	- Maintenance release. - Added <i>Taking the SPI Out of Reset</i> section.
2014.06.30	"Glue Logic for Master Port ss_in_n" section added - Interface Pins topic added - FPGA Routing topic added - Added address map and register descriptions
2014.02.28	Maintenance release.
2013.12.30	Minor formatting updates.
1.2	Minor updates.
1.1	Added programming model, address map and register definitions, clocks, and reset sections.
1.0	Initial release.

SPI Controller on page 19-1

Document Version	Changes
2015.05.04	- Added <i>Impact of SCL Rise Time and Fall Time On Generated SCL</i> figure to "Clock Synchronization" section - Updated "Minimum High and Low Counts" section
2014.12.15	- Maintenance release. - Added <i>Taking the I²C Out of Reset</i> section.
2014.06.30	HPS I ² C Signals for FPGA Routing table updated I ² C interface in FPGA Fabric diagram added Added Address Map and Register Descriptions
2014.02.28	Maintenance release.

Document Version	Changes
2013.12.30	Added HPS I ² c Signals for FPGA routing to "Interface Pins" section.
1.2	Minor updates.
1.1	Added programming model, address map and register definitions, clocks, reset, and interface pins sections.
1.0	Initial release.

I²C Controller on page 20-1

Document Version	Changes
2015.11.02	Renamed <i>Interface Pins</i> section to <i>HPS I/O Pins</i> and moved this section and <i>FPGA Routing</i> under <i>UART Controller Signal Description</i>
2015.05.04	Maintenance release.
2014.12.15	• Maintenance release. • Added <i>Taking the UART Out of Reset</i> section.
2014.06.30	• UART(RS232) Serial Protocol topic added • Interrupts section updated • Updated Interrupt type table • Added address map and register descriptions
2014.02.28	Maintenance release
2013.12.30	Minor formatting updates.
1.2	Minor updates.
1.1	Added programming model, address map and register definitions, and reset sections.
1.0	Initial release.

UART Controller on page 21-1

Document Version	Changes
2020.09.03	Added information about the state of HPS GPIO during cold reset in the <i>Taking the GPIO Interface Out of Reset</i> section.
2019.06.14	Added <i>GPIO State During Reset</i> section.

Document Version	Changes
2014.12.15	• Maintenance release. • Added <i>Taking the GPIO Out of Reset</i> section.
2014.06.30	Added Address Map and Register Descriptions
2014.02.28	Updated content in sections: • Features of the GPIO Interface • GPIO Interface Block Diagram and System Integration • Debounce Operation
2013.12.30	Minor formatting updates Updated GPIO interface block diagram and GPIO interface pin table
1.2	Minor updates.
1.1	Added programming model section.
1.0	Initial release.

General-Purpose I/O Interface on page 22-1

Document Version	Changes
2014.06.30	• "FPGA Interface" section added • Added address map and register descriptions
2014.02.28	Maintenance release.
2013.12.30	Minor formatting updates.
1.2	Minor updates.
1.1	Added programming model and address map and register definitions sections.
1.0	Initial release.

Timer on page 23-1

Document Version	Changes
2015.11.02	Added note to "Watchdog Timer Counter" section.

Document Version	Changes
2015.05.04	Maintenance release.
2014.12.15	• Maintenance release. • Added "Taking the Watchdog Timer Out of Reset" section.
2014.06.30	• "FPGA Interface" section added • Added address map and register descriptions
2014.02.28	Maintenance release.
2013.12.30	Minor formatting updates.
1.2	Minor updates.
1.1	Added programming model and address map and register definitions sections.
1.0	Initial release.

Watchdog Timer on page 24-1

Document Version	Changes
2013.12.30	Maintenance release
1.0	Maintenance release.
0.1	Preliminary draft.

Introduction to the HPS Component on page 25-1

Document Version	Changes
2015.11.02	Updated Sections: • Configuring Peripherals • Peripheral Signals Routed to FPGA
2015.05.04	Maintenance release.
2014.12.15	Maintenance release.
2014.06.30	• Updated the "Instantiating the HPS Component" section. • Added the "Using Unassigned I/O as Loan I/O" section. • Removed reference to CAN interrupts from the "Interrupts" section.

Document Version	Changes
2014.02.28	• Add interfaces to tables • Add parameters to General Parameters table
1.2	Maintenance release.
1.1	• Added debug interfaces • Added boot options • Corrected slave address width • Corrected SDRAM interface widths • Added TPIU peripheral • Added .sdc file generation • Added .tcl script for memory assignments
1.0	Initial release.
0.1	Preliminary draft.

Instantiating the HPS Component on page 26-1

Document Version	Changes
2015.05.04	• Added note to FGPA-to-HPS SDRAM Interface section • Added note to User Clocks section
2014.12.15	User Clock 2 has been removed
2014.06.30	Added address map and register descriptions

Document Version	Changes
2014.02.28	Added sections: • Peripheral FPGA Clocks • Peripheral Reset Interfaces • Boot from FPGA Interface • Input-only General Purpose Interfaces Removed section: • General Purpose Interfaces Updated sections: • Trace Port Interface Unit • Peripheral Signal Interfaces • FPGA-to-HPS Interrupts
2013.12.30	Minor formatting issues
1.1	• Added debug interfaces. • Updated HPS-to-FPGA reset interface names. • Updated HPS external reset source interface names. • Removed DMA peripheral interface clocks. • Referred to Address Span Extender.
1.0	Initial release.
0.1	Preliminary draft.

HPS Component Interfaces on page 27-1

Document Version	Changes
2016.05.03	Removed references to FPGA to HPS SDRAM simulation.
2015.11.02	Maintenance release
2015.05.04	Maintenance release
2014.12.15	Maintenance release
2014.11.14	• Updated the "Simulation Flows" section. • Updated the "Generating HPS Simulation Model in Platform Designer (Standard)" section.

Document Version	Changes
2014.02.28	Added new sections: • Boot from FPGA Interface • General Purpose Input (GPI) Interface Updated content in sections: • Specifying HPS Simulation Model in Platform Designer (Standard) • Running HPS RTL Simulation
2013.12.30	Maintenance release
1.1	• Added debug APB*, STM hardware event, FPGA cross trigger, FPGA trace port interfaces. • Added support for post-fit simulation. • Updated some API function names. • Removed DMA peripheral clock.
1.0	Initial release.
0.1	Preliminary draft.

Simulating the HPS Component on page 28-1

Document Version	Changes
2020.09.03	Added information about where the HPS IO Configuration is contained in the <i>Typical Preloader Boot Flow</i> section.
2018.07.17	• Removed <i>I/O State</i> subsection in <i>I/O Configuration</i> section. This content only applies to Intel® Arria® 10 HPS. • Added state of dedicated I/O at power up in the <i>I/O Configuration</i> section.
2016.05.27	Changed the name of the internal QSPI reference clock from <code>qspi_clk</code> to <code>qspi_ref_clk</code> ; and the external QSPI output clock, from <code>sclk_out</code> to <code>qspi_clk</code> .
2016.05.03	• Updated SD/MMC device clock values in the <i>CSEL Settings for SD/MMC Controller</i> section. • Included read capture delay information in the <i>Quad SPI Flash Delay Configuration</i> section. • Added bus mode to the "SD/MMC Controller Default Settings" table in the <i>Default Settings of the SD/MMC Controller</i> section. • Changed the name of the internal QSPI reference clock from <code>qspi_clk</code> to <code>qspi_ref_clk</code>; and the external QSPI output clock, from <code>sclk_out</code> to <code>qspi_clk</code>.
2015.11.02	Maintenance Release

Document Version	Changes
2015.05.04	- Added "Boot Source I/O Mapping" section - Removed "*2" multiplier in CSEL3 column of the table in "CSEL Pin Settings for the SD/MMC Controller" section. - Corrected frequency values for Device frequency and controller clock in the "NAND Controller CSEL Pin Settings" table in the "CSEL Settings for the NAND Controller" section - Removed reference to Mode Reset Command in the "Quad SPI Flash Devices" - Clarified "Shared Memory Locations" in "Shared Memory" section
2014.12.15	- Added the following sections: "Boot Overview" "FPGA Configuration Overview" "Booting and Configuration Options" "Boot Definitions" section with subsections on "Reset", "Boot ROM", "Boot Source I/O Pins", "Flash Memory Devices", "Clock Select", "I/O Configuration", "L4 Watchdog 0 Timer", "Preloader", and "U-Boot Loader". - Removed "Shared Memory" section
2014.06.30	Maintenance release
2014.02.28	Correction to "Leading the Preloader" section
2013.12.30	- Updated figures in the Booting and Configuration Introduction section. - Updated the Rest and Boot ROM sections. - Updated the Shared Memory Block table. - Updated register names in the Full Configuration section.
1.3	- Expanded shared memory block table. - Added CLKSEL tables. - Additional minor updates.
1.2	Updated the HPS boot and FPGA configuration sections.
1.1	- Updated the HPS boot section. - Added information about the flash devices used for HPS boot. - Added information about the FPGA configuration mode.
1.0	Initial release.

Booting and Configuration on page 29-1

2021.05.07

av_5v4



Subscribe



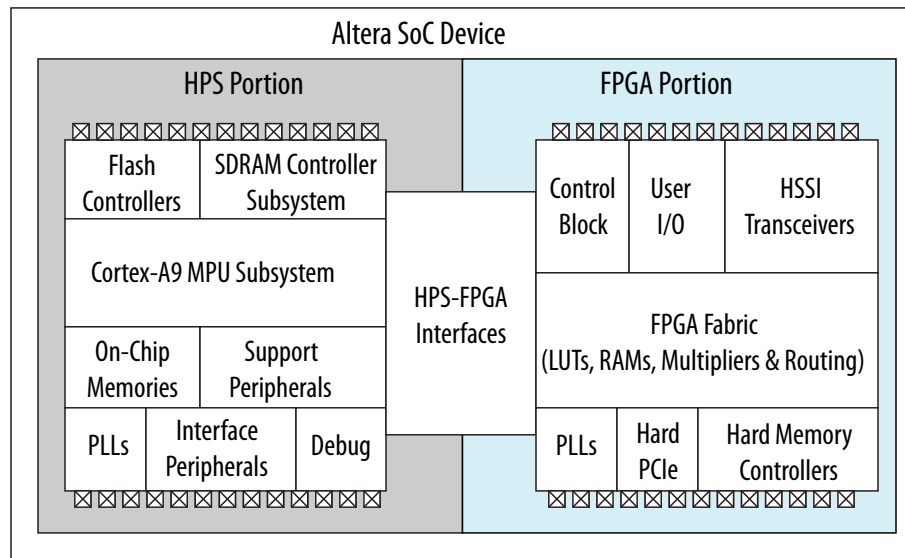
Send Feedback

The Arria V system-on-a-chip (SoC) is composed of two distinct portions- a dual-core Arm Cortex-A9 hard processor system (HPS) and an FPGA. The HPS architecture integrates a wide set of peripherals that reduce board size and increase performance within a system.

The SoC features the FPGA I/O, which is I/O pins dedicated to the FPGA fabric.

Figure 2-1: Intel SoC Device Block Diagram

Blocks connected to device pins have symbols (square with an X) adjacent to them in the figure.



The HPS consists of the following types of modules:

- Microprocessor unit (MPU) subsystem with a dual Arm Cortex-A9 MPCore processor
- Flash memory controllers
- SDRAM controller subsystem
- System interconnect
- On-chip memories
- Support peripherals
- Interface peripherals
- Debug components
- Phase-locked loops (PLLs)

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

The HPS incorporates third-party intellectual property (IP) from several vendors.

The dual-processor HPS supports symmetric (SMP) and asymmetric (AMP) multiprocessing.

The FPGA portion of the device contains:

- FPGA fabric
- Control block (CB)
- PLLs
- High-speed serial interface (HSSI) transceivers, depending on the device variant
- Hard PCI Express® (PCI-e) controllers
- Hard memory controllers

The HPS and FPGA communicate with each other through bus interfaces that bridge the two distinct portions. On a power-on reset, the HPS can boot from multiple sources, including the FPGA fabric and external flash. The FPGA can be configured through the HPS or an externally supported device.

The HPS and FPGA portions of the device each have their own pins. Pins are not freely shared between the HPS and the FPGA fabric. The HPS I/O pins are configured by boot software executed by the MPU in the HPS. Software executing on the HPS accesses control registers in the system manager to assign HPS I/O pins to the available HPS modules. The FPGA I/O pins are configured by an FPGA configuration image through the HPS or any external source supported by the device.

The MPU subsystem can boot from flash devices connected to the HPS pins. When the FPGA portion is configured by an external source, the MPU subsystem can boot from flash memory devices available to the FPGA portion of the device.

The HPS and FPGA portions of the device have separate external power supplies and independently power on. You can power on the HPS without powering on the FPGA portion of the device. However, to power on the FPGA portion, the HPS must already be on or powered at the same time as the FPGA portion. You can also turn off the FPGA portion of the device while leaving the HPS power on.

Table 2-1: Valid SoC Power Modes

HPS	FPGA	Valid?
Off	Off	Yes
Off	On	No
On	Off	Yes
On	On	Yes

Related Information

- [Booting and Configuration](#) on page 29-1
- [Arria V Device Datasheet](#)

Features of the HPS

The main modules of the HPS are:

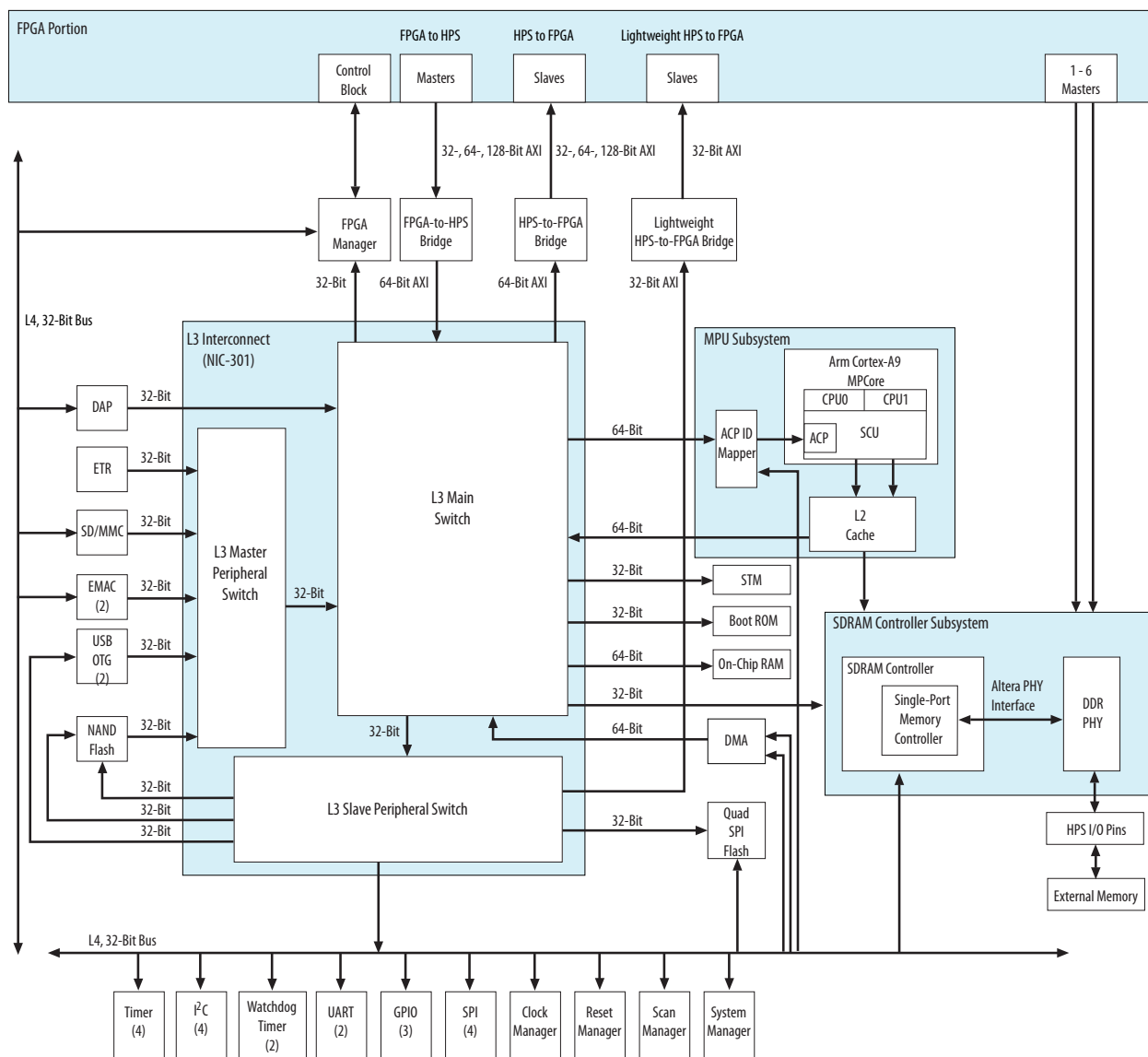
- MPU subsystem featuring a dual-core Arm Cortex-A9 MPCore processor
- General-purpose direct memory access (DMA) controller
- Two Ethernet media access controllers (EMACs)
- Two USB 2.0 on-the-go (OTG) controllers
- NAND flash controller
- Quad SPI flash controller
- Secure digital/multimedia card (SD/MMC) controller
- Two serial peripheral interface (SPI) master controllers
- Two SPI slave controllers
- Four inter-integrated circuit (I²C) controllers⁽¹⁾
- 64 KB on-chip RAM
- 64 KB on-chip boot ROM
- Two UARTs
- Four timers
- Two watchdog timers
- Three general-purpose I/O (GPIO) interfaces
- Arm CoreSight debug components:
 - Debug access port (DAP)
 - Trace port interface unit (TPIU)
 - System trace macrocell (STM)
 - Program trace macrocell (PTM)
 - Embedded trace router (ETR)
 - Embedded cross trigger (ECT)
- System manager
- Clock manager
- Reset manager
- Scan manager
- FPGA manager
- SDRAM controller subsystem

⁽¹⁾ Two of the four I²Cs, provide support for general purpose.

HPS Block Diagram and System Integration

HPS Block Diagram

Figure 2-2: HPS Block Diagram



Cortex-A9 MPCore

The MPU subsystem provides the following functionality:

- Arm Cortex-A9 MPCore
 - Two Arm Cortex-A9 processors
 - NEON™ single instruction, multiple data (SIMD) coprocessor and vector floating-point v3 (VFPv3) per processor
 - Snoop control unit (SCU) to ensure coherency
 - Accelerator coherency port (ACP) that accepts coherency memory access requests
 - Interrupt controller
 - One general-purpose timer and one watchdog timer per processor
 - Debug and trace features
 - 32 KB instruction and 32 KB data level 1 (L1) caches per processor
 - Memory management unit (MMU) per processor
- Arm L2-310 level 2 (L2) cache
 - Shared 512 KB L2 cache
- ACP ID mapper
 - Maps the 12-bit ID from the level 3 (L3) interconnect to the 3-bit ID supported by the ACP

A programmable address filter in the L2 cache controls which portions of the 32-bit physical address space can be accessed by each master.

Related Information

[HPS Block Diagram](#) on page 2-4

HPS Interfaces

The Arria V device family provides multiple communication channels to the HPS.

HPS-FPGA Memory-Mapped Interfaces

The HPS-FPGA memory-mapped interfaces provide the major communication channels between the HPS and the FPGA fabric. The HPS-FPGA memory-mapped interfaces include:

- FPGA-to-HPS bridge—a high-performance bus with a configurable data width of 32, 64, or 128 bits, allowing the FPGA fabric to master transactions to the slaves in the HPS. This interface allows the FPGA fabric to have full visibility into the HPS address space. This interface also provides access to the coherent memory location
- HPS-to-FPGA bridge—a high-performance interface with a configurable data width of 32, 64, or 128 bits, allowing the HPS to master transactions to slaves in the FPGA fabric
- Lightweight HPS-to-FPGA bridge—an interface with a 32-bit fixed data width, allowing the HPS to master transactions to slaves in the FPGA fabric. This lower-bandwidth interface is useful for accessing the control and status registers of soft peripherals

Related Information

[HPS-FPGA Bridges](#) on page 9-1

Other HPS Interfaces

- TPIU trace—sends trace data created in the HPS to the FPGA fabric
- FPGA System Trace Macrocell (STM) —an interface that allows the FPGA fabric to send hardware events to be stored in the HPS trace data
- FPGA cross-trigger—an interface that allows the CoreSight trigger system to send triggers to IP cores in the FPGA, and vice versa
- DMA peripheral interface—multiple peripheral-request channels
- FPGA manager interface—signals that communicate with the FPGA fabric for boot and configuration
- Interrupts—allow soft IP cores to supply interrupts directly to the MPU interrupt controller
- MPU standby and events—signals that notify the FPGA fabric that the MPU is in standby mode and signals that wake up Cortex-A9 processors from a wait for event (WFE) state
- HPS debug interface – an interface that allows the HPS debug control domain (debug APB) to extend into FPGA

Other HPS–FPGA communications channels:

- FPGA clocks and resets
- HPS-to-FPGA JTAG—allows the HPS to master the FPGA JTAG chain

System Interconnect

The system interconnect consists of the main L3 interconnect and level 4 (L4) buses. The L3 interconnect is an Arm NIC-301 module composed of the following switches:

- L3 main switch
 - Connects the master, slaves, and other subswitches
 - Provides 64-bit switching capabilities
- L3 master peripheral switch
 - Connects master ports of peripherals with integrated DMA controllers to the L3 main switch
- L3 slave peripheral switch
 - Connects slave ports of peripherals to the L3 main switch

Related Information

[System Interconnect](#) on page 8-1

SDRAM Controller Subsystem

HPS and FPGA fabric masters have access to the SDRAM controller subsystem.

The SDRAM controller subsystem implements the following high-level features:

- Support for double data rate 2 (DDR2), DDR3, and low-power double data rate 2 (LPDDR2) devices
- Error correction code (ECC) support, including calculation, single-bit error correction and write-back, and error counters
- Fully-programmable timing parameter support for all JEDEC-specified timing parameters
- All ports support memory protection and mutual accesses
- FPGA fabric interface with up to six ports that can be combined for a data width up to 256-bits wide using Avalon-MM and AXI interfaces.

The SDRAM controller subsystem is composed of the SDRAM controller, DDR PHY, control and status registers and their associated interfaces.

Related Information

[SDRAM Controller Subsystem](#) on page 11-1

SDRAM Controller

The SDRAM controller contains a multiport front end (MPFE) that accepts requests from HPS masters and from soft logic in the FPGA fabric through the FPGA-to-HPS SDRAM interface.

The SDRAM controller offers the following features:

- Up to 4 GB address range
- 8-, 16-, and 32-bit data widths
- Optional ECC support
- Low-voltage 1.35V DDR3L and 1.2V DDR3U support
- Full memory device power management support
- Two chip selects (DDR2 and DDR3)

The SDRAM controller provides the following features to maximize memory performance:

- Command reordering (look-ahead bank management)
- Data reordering (out of order transactions)
- Priority arbitration and deficit round-robin arbitration for ports with the same priority
- High-priority bypass for latency sensitive traffic

Related Information

[SDRAM Controller Subsystem](#) on page 11-1

DDR PHY

The DDR PHY interfaces the single port memory controller to the HPS memory I/O.

Related Information

[SDRAM Controller Subsystem](#) on page 11-1

On-Chip Memory

On-Chip RAM

The on-chip RAM offers the following features:

- 64 KB size
- 64-bit slave interface
- High performance for all burst lengths

Related Information

[On-Chip Memory](#) on page 12-1

Boot ROM

The boot ROM offers the following features:

- 64 KB size
- Contains the code required to support HPS boot from cold or warm reset
- Used exclusively for booting the HPS

Related Information

- [On-Chip Memory](#) on page 12-1
- [Booting and Configuration](#) on page 29-1

Flash Memory Controllers

NAND Flash Controller

The NAND flash controller is based on the Cadence® Design IP® NAND Flash Memory Controller and offers the following functionality and features:

- Supports one x8 NAND flash device
- Supports Open NAND Flash Interface (ONFI) 1.0
- Supports NAND flash memories from Hynix, Samsung, Toshiba, Micron, and ST Micro
- Supports programmable 512 byte (4-, 8-, or 16-bit correction) or 1024 byte (24-bit correction) ECC sector size
- Supports pipeline read-ahead and write commands for enhanced read/write throughput
- Supports devices with 32, 64, 128, 256, 384, or 512 pages per block
- Supports multiplane devices
- Supports page sizes of 512 bytes, 2 kilobytes (KB), 4 KB, or 8 KB
- Supports single-level cell (SLC) and multi-level cell (MLC) devices with programmable correction capabilities
- Provides internal DMA
- Provides programmable access timing

Related Information

[NAND Flash Controller](#) on page 13-1

Quad SPI Flash Controller

The quad SPI flash controller is based on the Cadence Quad SPI Flash Controller and offers the following features:

- Supports SPIx1, SPIx2, or SPIx4 (Quad SPI) serial NOR flash devices
- Supports direct access and indirect access modes
- Supports single, dual, and quad I/O instructions
- Support up to four chip selects
- Programmable write-protected regions
- Programmable delays between transactions
- Programmable device sizes
- Support eXecute-In-Place (XIP) mode
- Programmable baud rate generator to generate device clocks

Related Information

[Quad SPI Flash Controller](#) on page 15-1

SD/MMC Controller

The Secure Digital (SD), Multimedia Card (MMC), (SD/MMC) and CE-ATA host controller is based on the Synopsys DesignWare* Mobile Storage Host controller and offers the following features:

- Integrated descriptor-based DMA
- Supports CE-ATA digital protocol commands
- Supports single card
- Single data rate (SDR) mode only
- Programmable card width: 1-, 4-, and 8-bit
- Programmable card types: SD, SDIO, or MMC
- Up to 64 KB programmable block size
- Supports the following standards and card types:
 - SD, including eSD—version 3.0⁽²⁾
 - SDIO, including embedded SDIO (eSDIO)—version 3.0⁽³⁾
 - CE-ATA—version 1.1
- Supports various types of multimedia cards, MMC version 4.41⁽⁴⁾
 - MMC: 1-bit data bus
 - Reduced-size MMC (RSMMC): 1-bit and 4-bit data bus
 - MMCMobile: 1-bit data bus
 - MMCPlus: 1-bit, 4-bit, and 8-bit data bus
 - Default speed and high speed
- Supports embedded MMC (eMMC) version 4.41⁽⁵⁾
 - 1-bit, 4-bit and 8-bit data bus

Note: For an inclusive list of the programmable card types and versions supported, refer to the *SD/MMC Controller* chapter.

Related Information

[SD/MMC Controller](#) on page 14-1

Support Peripherals

Clock Manager

The clock manager offers the following features:

- Manages clocks for HPS
- Supports dynamic clock tuning

Related Information

[Clock Manager](#) on page 3-1

⁽²⁾ Does not support SDR50, SDR104, and DDR50 modes.

⁽³⁾ Does not support SDR50, SDR104, and DDR50 modes.

⁽⁴⁾ Does not support DDR mode.

⁽⁵⁾ Does not support DDR mode.

Reset Manager

The reset manager manages both hardware and software reset sources in the HPS. Reset status is also provided. Reset types include cold, warm, and debug. Reset behavior depends on the type.

Related Information

[Reset Manager](#) on page 4-1

System Manager

The system manager controls system functions and modules that need external control signals. The system manager offers the following features:

- ECC monitoring and control
- Low-level control of peripheral features not accessible through the control and status registers (CSRs)
- Freeze controller that places I/O elements into a safe state for configuration

Related Information

[System Manager](#) on page 6-1

Scan Manager

The scan manager is used to configure and manage HPS I/O pins and to communicate with the FPGA JTAG.

Related Information

[Scan Manager](#) on page 7-1

Timers

The four timers are based on the Synopsys DesignWare APB Timer peripheral and offer the following features:

- 32-bit timer resolution
- Free-running timer mode
- Supports a time-out period of up to 43 seconds when the timer clock frequency is 100 MHz
- Interrupt generation

Related Information

[Timer](#) on page 23-1

Watchdog Timers

The two watchdog timers are based on the Synopsys DesignWare APB Watchdog Timer peripheral and offer the following features:

- 32-bit timer resolution
- Interrupt request
- Reset request
- Programmable time-out period up to approximately 86 seconds (assuming a 50 MHz input clock frequency)

Related Information

[Watchdog Timer](#) on page 24-1

DMA Controller

The DMA controller provides high-bandwidth data transfers for modules without integrated DMA controllers. The DMA controller is based on the Arm Corelink* DMA Controller (DMA-330) and offers the following features:

- Micro-coded to support flexible transfer types
 - Memory-to-memory
 - Memory-to-peripheral
 - Peripheral-to-memory
 - Scatter-gather
- Supports up to eight channels
- Supports flow control with 31 peripheral handshake interfaces
- Software can schedule up to 16 outstanding read and 16 outstanding write instructions
- Supports nine interrupt lines: one for DMA thread abort and eight for external events

Related Information

[DMA Controller](#) on page 16-1

FPGA Manager

The FPGA manager offers the following features:

- Manages configuration of the FPGA portion of the device
- 32-bit fast passive parallel configuration interface to the FPGA CSS block
- Partial reconfiguration
- Compressed FPGA configuration images
- Advanced Encryption Standard (AES) encrypted FPGA configuration images
- Monitors configuration-related signals in FPGA
- Provides 32 general-purpose inputs and 32 general-purpose outputs to the FPGA fabric

Related Information

[FPGA Manager](#) on page 5-1

Interface Peripherals

EMACs

The two EMACs are based on the Synopsys DesignWare 3504-0 Universal 10/100/1000 Ethernet MAC and offer the following features:

- Supports 10, 100, and 1000 Mbps standard
- Supports RGMII external PHY interface
 - Media independent interface (MII)
 - Gigabit media independent interface (GMII)
 - Reduced gigabit media independent interface (RGMII)
 - Serial gigabit media independent interface (SGMII) with additional external conversion logic
- Provides full GMII interface when using FPGA interface
- Integrated DMA controller
- Supports IEEE 1588-2002 and IEEE 1588-2008 standards for precision networked clock synchronization

- IEEE 802.3-az, version D2.0 of Energy Efficient Ethernet
- Supports IEEE 802.1Q VLAN tag detection for reception frames
- Supports a variety of address filtering modes
- Management of PHY through Management data input/output (MDIO) interface or optionally, I²C interface

Related Information

[Ethernet Media Access Controller](#) on page 17-1

USB Controllers

The HPS provides two USB 2.0 Hi-Speed On-the-Go (OTG) controllers from Synopsys DesignWare. The USB controller signals cannot be routed to the FPGA like those of other peripherals; instead they are routed to the dedicated I/O.

Each of the USB controllers offers the following features:

- Complies with the following specifications:
 - USB OTG Revision 1.3
 - USB OTG Revision 2.0
 - Embedded Host Supplement to the USB Revision 2.0 Specification
- Supports software-configurable modes of operation between OTG 1.3 and OTG 2.0
- Supports all USB 2.0 speeds:
 - High speed (HS, 480-Mbps)
 - Full speed (FS, 12-Mbps)
 - Low speed (LS, 1.5-Mbps)

Note: In host mode, all speeds are supported; however, in device mode, only high speed and full speed are supported.
- Local buffering with Error Correction Code (ECC) support

Note: The USB 2.0 OTG controller does not support the following interface standards:

 - Enhanced Host Controller Interface (EHCI)
 - Open Host Controller Interface (OHCI)
 - Universal Host Controller Interface (UHCI)
- Supports USB 2.0 Transceiver Macrocell Interface Plus (UTMI+) Low Pin Interface (ULPI) PHYs (SDR mode only)
- Supports up to 16 bidirectional endpoints, including control endpoint 0

Note: Only seven periodic device IN endpoints are supported.
- Supports up to 16 host channels

Note: In host mode, when the number of device endpoints is greater than the number of host channels, software can reprogram the channels to support up to 127 devices, each having 32 endpoints (IN + OUT), for a maximum of 4,064 endpoints.
- Supports generic root hub
- Supports automatic ping capability

Related Information

- [USB 2.0 OTG Controller](#) on page 18-1

- **Universal Serial Bus (USB) website**

Additional information is available in the On The Go and Embedded Host Supplement to the USB Revision 2.0 Specification, which you can download from the USB Implementers Forum website.

I²C Controllers

The four I²C controllers are based on Synopsys DesignWare APB I²C controller which offer the following features:

- Two controllers support I²C management interfaces for use by the EMAC controllers
- Support both 100 Kbps and 400 Kbps modes
- Support both 7-bit and 10-bit addressing modes
- Support master and slave operating mode
- Direct access for host processor
- DMA controller may be used for large transfers

Related Information

[I²C Controller](#) on page 20-1

UARTs

The HPS provides two UART controllers to provide asynchronous serial communications. The two UART modules are based on Synopsys DesignWare APB Universal Asynchronous Receiver/ Transmitter peripheral and offer the following features:

- 16550-compatible UART
- Support automatic flow control as specified in 16750 standard
- Programmable baud rate up to 6.25 MBaud (with 100MHz reference clock)
- Direct access for host processor
- DMA controller may be used for large transfers
- 128-byte transmit and receive FIFO buffers

Related Information

[UART Controller](#) on page 21-1

SPI Master Controllers

The two SPI master controllers are based on Synopsys DesignWare Synchronous Serial Interface (SSI) controller and offer the following features:

- Choice of Motorola* SPI, Texas Instruments* Synchronous Serial Protocol or National Semiconductor* Microwire protocol
- Programmable data frame size from 4 bits to 16 bits
- Supports full- and half-duplex modes
- Supports up to four chip selects
- Direct access for host processor
- DMA controller may be used for large transfers
- Programmable master serial bit rate
- Support for rxd sample delay
- Transmit and receive FIFO buffers are 256 words deep

Related Information

[SPI Controller](#) on page 19-1

SPI Slave Controllers

The two SPI slave controllers are based on Synopsys DesignWare Synchronous Serial Interface (SSI) controller and offer the following features:

- Programmable data frame size from 4 bits to 16 bits
- Supports full- and half-duplex modes
- Direct access for host processor
- DMA controller may be used for large transfers
- Transmit and receive FIFO buffers are 256 words deep

Related Information

[SPI Controller](#) on page 19-1

GPIO Interfaces

The HPS provides three GPIO interfaces that are based on Synopsys DesignWare APB General Purpose Programming I/O peripheral and offer the following features:

- Supports digital de-bounce
- Configurable interrupt mode
- Supports up to 71 I/O pins and 14 input-only pins, based on device variant

Related Information

[General-Purpose I/O Interface](#) on page 22-1

CoreSight Debug and Trace

The CoreSight debug and trace system offers the following features:

- Real-time program flow instruction trace through a separate PTM for each processor
- Host debugger JTAG interface
- Connections for cross-trigger and STM-to-FPGA interfaces, which enable soft IP cores to generate of triggers and system trace messages
- Custom message injection through STM into trace stream for delivery to host debugger
- Capability to route trace data to any slave accessible to the ETR master, which is connected to the level 3 (L3) interconnect

Related Information

[CoreSight Debug and Trace](#) on page 10-1

Endian Support

The HPS is natively a little-endian system. All HPS slaves are little endian.

The processor masters are software configurable to interpret data as little endian, big endian, or byte-invariant (BE8). All other masters, including the USB 2.0 interface, are little endian. Registers in the MPU and L2 cache are little endian regardless of the endian mode of the CPUs.

Note: Intel strongly recommends that you only use little endian.

The FPGA-to-HPS, HPS-to-FPGA, FPGA-to-SDRAM, and lightweight HPS-to-FPGA interfaces are little endian.

If a processor is set to BE8 mode, software must convert endianness for accesses to peripherals and DMA linked lists in memory. The processor provides instructions to swap byte lanes for various sizes of data.

The Arm Cortex-A9 MPU supports a single instruction to change the endianness of the processor and provides the REV and REV16 instructions to swap the endianness of bytes or half-words respectively. The MMU page tables are software configurable to be organized as little-endian or BE8.

The Arm DMA controller is software configurable to perform byte lane swapping during a transfer.

Introduction to the Hard Processor System Address Map

The address map specifies the addresses of slaves, such as memory and peripherals, as viewed by the MPU and other masters. The HPS has multiple address spaces, defined in the following section.

Related Information
[System Interconnect](#) on page 8-1

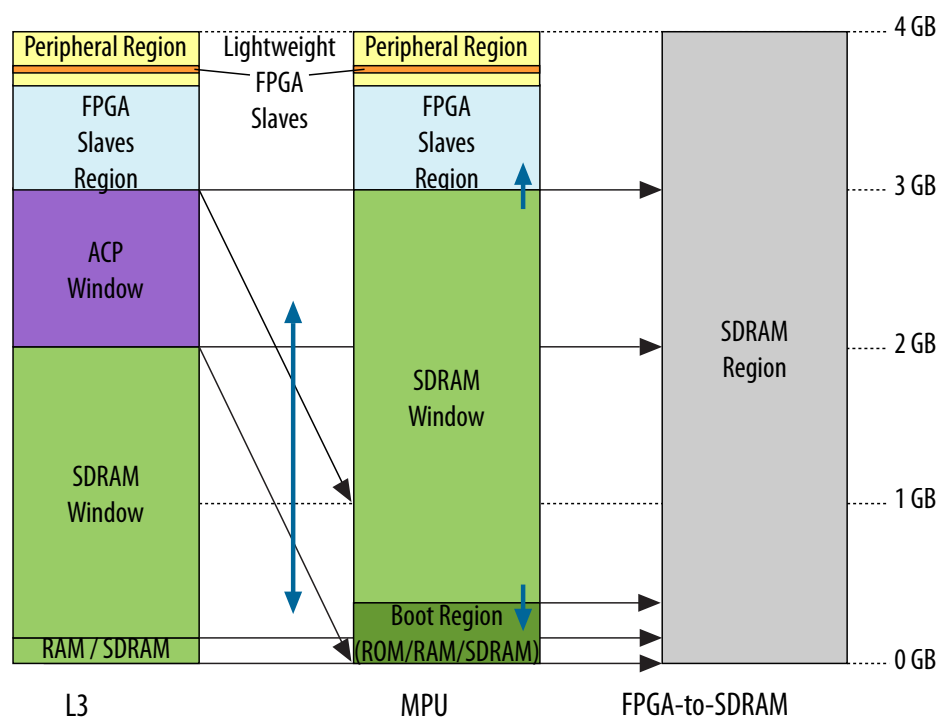
HPS Address Spaces

The following table shows the HPS address spaces and their sizes.

Address spaces are divided into one or more nonoverlapping regions. For example, the MPU address space has the peripheral, FPGA slaves, SDRAM window, and boot regions.

The following figure shows the relationships between the HPS address spaces. The figure is not to scale.

Figure 2-3: HPS Address Space Relationships



The window regions provide access to other address spaces. The thin black arrows indicate which address space is accessed by a window region (arrows point to accessed address space). For example, accesses to the ACP window in the L3 address space map to a 1 GB region of the MPU address space.

The SDRAM window in the MPU address space can grow and shrink at the top and bottom (short, blue vertical arrows) at the expense of the FPGA slaves and boot regions. For specific details, refer to “MPU Address Space”.

The ACP window can be mapped to any 1 GB region in the MPU address space (blue vertical bidirectional arrow), on gigabyte-aligned boundaries.

The following table shows the base address and size of each region that is common to the L3 and MPU address spaces.

Table 2-2: Common Address Space Regions

Region Name	Base Address	Size
FPGA slaves	0xC0000000	960 MB
Peripheral	0xFC000000	64 MB
Lightweight FPGA slaves	0xFF200000	2 MB

SDRAM Address Space

The SDRAM address space is up to 4 GB. The entire address space can be accessed through the FPGA-to-HPS SDRAM interface from the FPGA fabric. The total amount of SDRAM addressable from the other address spaces can be configured at runtime.

Related Information

[System Interconnect](#) on page 8-1

For more information about how to configure SDRAM address space.

MPU Address Space

The MPU address space is 4 GB and applies to addresses generated inside the MPU.

The MPU address space contains the following regions:

- The SDRAM window region provides access to a large, configurable portion of the 4 GB SDRAM address space.

The address filtering start and end registers in the L2 cache controller define the SDRAM window boundaries. The boundaries are megabyte-aligned. Addresses within the boundaries route to the SDRAM master. Addresses outside the boundaries route to the system interconnect master.

Related Information

- [HPS Address Spaces](#) on page 2-15
For more information, refer to the "HPS Address Spaces Relationships" table.
- [System Interconnect](#) on page 8-1
For more information regarding SDRAM address mapping, refer to the System Interconnect chapter.
- [Cortex-A9 Microprocessor Unit Subsystem](#)

L3 Address Space

The L3 address space is 4 GB and applies to all L3 masters except the MPU. The L3 address space has more configuration options than the other address spaces.

Related Information

[System Interconnect](#) on page 8-1

For more information about configuring the L3 address space, refer to the System Interconnect chapter.

HPS Peripheral Region Address Map

Each peripheral slave interface has a dedicated address range in the peripheral region. The table below lists the base address and address range size for each slave.

Table 2-3: Peripheral Region Address Map

Slave Identifier	Description	Base Address	Size
STM	Space Trace Macrocell	0xFC000000	48 MB
DAP	Debug Access Port	0xFF000000	2 MB
LWFPGASLAVES	FPGA slaves accessed with lightweight HPS-to-FPGA bridge	0xFF200000	2 MB
LWHPS2FPGAREGS	Lightweight HPS-to-FPGA bridge global programmer's view (GPV) registers	0xFF400000	1 MB
HPS2FPGAREGS	HPS-to-FPGA bridge GPV registers	0xFF500000	1 MB
FPGA2HPSREGS	FPGA-to-HPS bridge GPV registers	0xFF600000	1 MB
EMAC0	Ethernet MAC 0	0xFF700000	8 KB
EMAC1	Ethernet MAC 1	0xFF702000	8 KB
SDMMC	SD/MMC	0xFF704000	4 KB
QSPIREGS	Quad SPI flash controller registers	0xFF705000	4 KB
FPGAMGRREGS	FPGA manager registers	0xFF706000	4 KB
ACPIDMAP	ACP ID mapper registers	0xFF707000	4 KB
GPIO0	GPIO 0	0xFF708000	4 KB

Slave Identifier	Description	Base Address	Size
GPIO1	GPIO 1	0xFF709000	4 KB
GPIO2	GPIO 2	0xFF70A000	4 KB
L3REGS	L3 interconnect GPV	0xFF800000	1 MB
NANDDATA	NAND flash controller data	0xFF900000	64 KB
QSPIDATA	Quad SPI flash data	0xFFA00000	1 MB
USB0	USB 2.0 OTG 0 controller registers	0xFFB00000	256 KB
USB1	USB 2.0 OTG 1 controller registers	0xFFB40000	256 KB
NANDREGS	NAND flash controller registers	0xFFB80000	64 KB
FPGAMGRDATA	FPGA manager configuration data	0xFFB90000	4 KB
UART0	UART 0	0xFFC02000	4 KB
UART1	UART 1	0xFFC03000	4 KB
I2C0	I ² C controller 0	0xFFC04000	4 KB
I2C1	I ² C controller 1	0xFFC05000	4 KB
I2C2	I ² C controller 2	0xFFC06000	4 KB
I2C3	I ² C controller 3	0xFFC07000	4 KB
SPTIMER0	SP Timer 0	0xFFC08000	4 KB
SPTIMER1	SP Timer 1	0xFFC09000	4 KB
SDRREGS	SDRAM controller subsystem registers	0xFFC20000	128 KB
OSC1TIMER0	OSC1 Timer 0	0xFFD00000	4 KB
OSC1TIMER1	OSC1 Timer 1	0xFFD01000	4 KB
L4WD0	Watchdog Timer 0	0xFFD02000	4 KB

Slave Identifier	Description	Base Address	Size
L4WD1	Watchdog Timer 1	0xFFD03000	4 KB
CLKMGR	Clock manager	0xFFD04000	4 KB
RSTMGR	Reset manager	0xFFD05000	4 KB
SYSMGR	System manager	0xFFD08000	16 KB
DMANONSECURE	DMA nonsecure registers	0xFFE00000	4 KB
DMASECURE	DMA secure registers	0xFFE01000	4 KB
SPIS0	SPI slave 0	0xFFE02000	4 KB
SPIS1	SPI slave 1	0xFFE03000	4 KB
SPIM0	SPI master 0	0xFFF00000	4 KB
SPIM1	SPI master 1	0xFFF01000	4 KB
SCANMGR	Scan manager registers	0xFFF02000	4 KB
ROM	Boot ROM	0xFFFD0000	64 KB
MPU	MPU registers	0xFFFEC000	8 KB
MPUL2	MPU L2 cache controller registers	0xFFFEF000	4 KB
OCRAM	On-chip RAM	0xFFFF0000	64 KB

Related Information**[Arria V Address Map and Register Definitions](#)**

Web-based address map and register definitions

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) clock generation is centralized in the clock manager. The clock manager is responsible for providing software-programmable clock control to configure all clocks generated in the HPS. Clocks are organized in clock groups. A clock group is a set of clock signals that originate from the same clock source. A phase-locked loop (PLL) clock group is a clock group where the clock source is a common PLL voltage-controlled oscillator (VCO).

Features of the Clock Manager

The *Clock Manager* offers the following features:

- Generates and manages clocks in the HPS
- Contains the following PLL clock groups:
 - PLL 0 (Main)—contains clocks for the Arm Cortex-A9 microprocessor unit (MPU) subsystem, level 3 (L3) interconnect, level 4 (L4) peripheral bus, and debug
 - PLL 1 (Peripheral)—contains clocks for PLL-driven peripherals
 - SDRAM—contains clocks for the SDRAM subsystem
- Allows scaling of the MPU subsystem clocks without disabling peripheral and SDRAM clock groups
- Generates clock gate controls for enabling and disabling most clocks
- Initializes and sequences clocks for the following events:
 - Cold reset
 - Safe mode request from reset manager on warm reset
- Allows software to program clock characteristics, such as the following items discussed later in this chapter:
 - Input clock source for SDRAM and peripheral PLLs
 - Multiplier range, divider range, and six post-scale counters for each PLL
 - Output phases for SDRAM PLL outputs
 - VCO enable for each PLL
 - Bypass modes for each PLL
 - Gate off individual clocks in all PLL clock groups
 - Clear loss of lock status for each PLL
 - Safe mode for hardware-managed clocks
 - General-purpose I/O (GPIO) debounce clock divide

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

- Allows software to observe the status of all writable registers
- Supports interrupting the MPU subsystem on PLL-lock and loss-of-lock
- Supports clock gating at the signal level

The clock manager is **not** responsible for the following functional behaviors:

- Selection or management of the clocks for the FPGA-to-HPS and HPS-to-FPGA interfaces. The FPGA logic designer is responsible for selecting and managing these clocks.
- Software must not program the clock manager with illegal values. If it does, the behavior of the clock manager is undefined and could stop the operation of the HPS. The only guaranteed means for recovery from an illegal clock setting is a cold reset.
- When re-programming clock settings, there are no automatic glitch-free clock transitions. Software must follow a specific sequence to ensure glitch-free clock transitions. Refer to *Hardware-Managed and Software-Managed Clocks* section of this chapter.

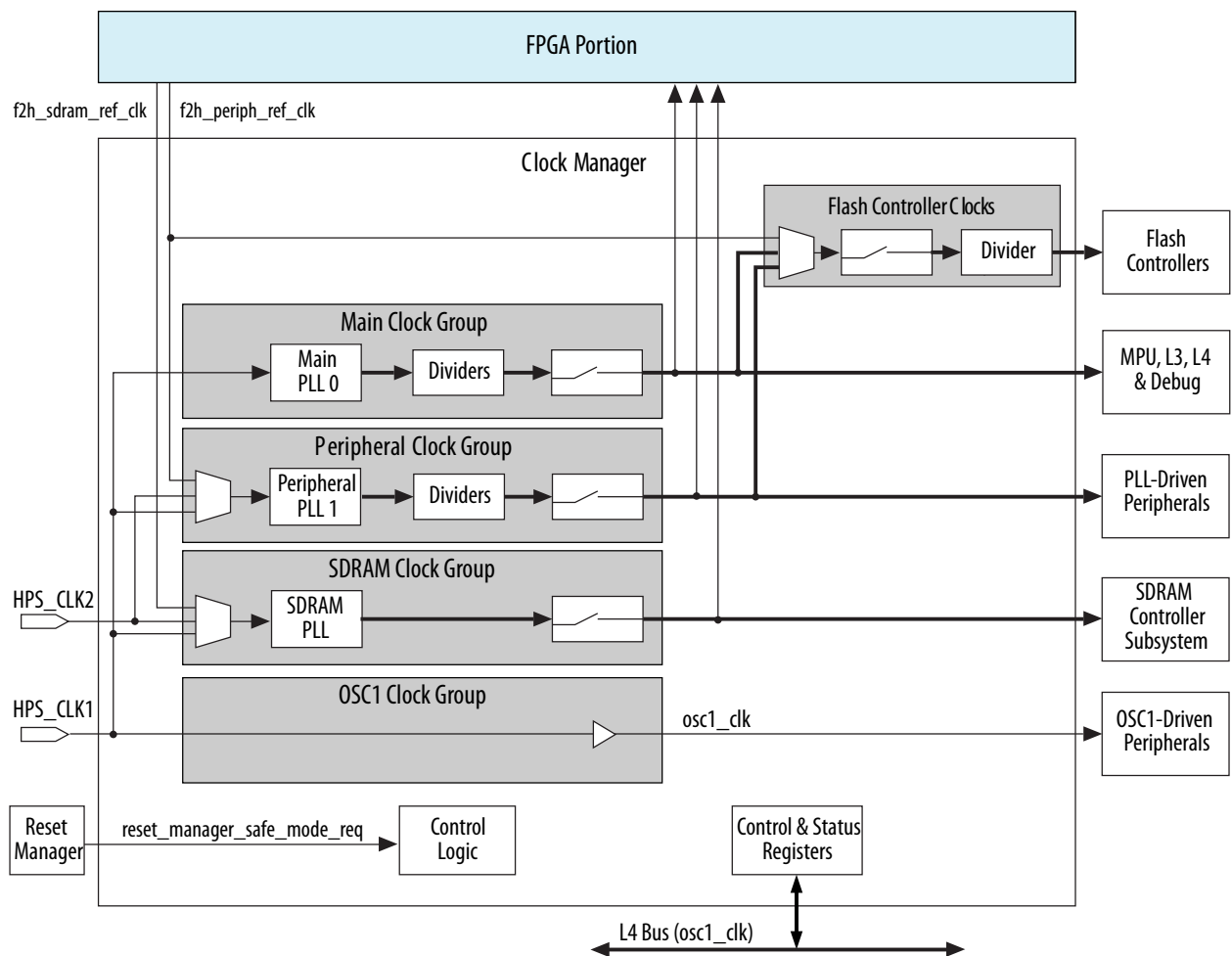
Related Information

[Hardware-Managed and Software-Managed Clocks](#) on page 3-7

Clock Manager Block Diagram and System Integration

Figure 3-1: Clock Manager Block Diagram

The following figure shows the major components of the clock manager and its integration in the HPS.



L4 Peripheral Clocks

The L4 peripheral clocks, denoted by `l4_mp_clk`, range up to 200 MHz.

Table 3-1: Clock List

Peripheral	Clock Name	Description
USB OTG 0/1 ⁽⁶⁾	hclk	AHB* clock
	pmu_hclk	PMU AHB clock. pmu_hclk is the scan clock for the PMU's AHB domain. Note: Select it as a test clock.
	utmi_clk	Always used as the PHY domain clock during DFT Scan mode. Note: Select utmi_clk as a test clock even when the core is configured for a non-UTMI PHY.
Quad SPI Flash Controller ⁽⁶⁾	pcclk	APB clock
	hclk	AHB clock
NAND Flash Controller (Locally gated nand_mp_clk.) ⁽⁶⁾	ACLK	AHB Data port clock
	mACLK	AXI* Master port clock
	regACLK	AHB Register port clock
	ecc_clk	ECC circuitry clock
	clk_x	Bus Interface Clock
EMAC 0/1	acclk	Application clock for DMA AXI bus and CSR APB bus.
SD/MMC Controller	sdmmc_clk	All registers reside in the BIU clock domain.

For more information about the specific peripheral clocks, refer to their respective chapters.

Related Information

- [SD/MMC Controller](#) on page 14-1
- [NAND Flash Controller](#) on page 13-1
- [Quad SPI Flash Controller](#) on page 15-1
- [Ethernet Media Access Controller](#) on page 17-1
- [USB 2.0 OTG Controller](#) on page 18-1

⁽⁶⁾ Clock manager provides CSR bits for software enables to some peripherals. These enables are defaulted to enable. In boot mode, these enables are automatically active to ensure all clocks are active if RAM is cleared for security.

Functional Description of the Clock Manager

Clock Manager Building Blocks

PLLs

The clock manager contains three PLLs: PLL 0 (main), PLL 1 (peripherals), and SDRAM. These PLLs generate the majority of clocks in the HPS. There is no phase control between the clocks generated by the three PLLs.

Each PLL has the following features:

- Phase detector and output lock signal generation
- Registers to set VCO frequency
 - (M) Multiplier range is 1 to 4096
 - (N) Divider range is 1 to 64
- Six post-scale counters (C0-C5) with a range of 1 to 512
- PLL can be enabled to bypass all outputs to the osc1_clk clock for glitch-free transitions

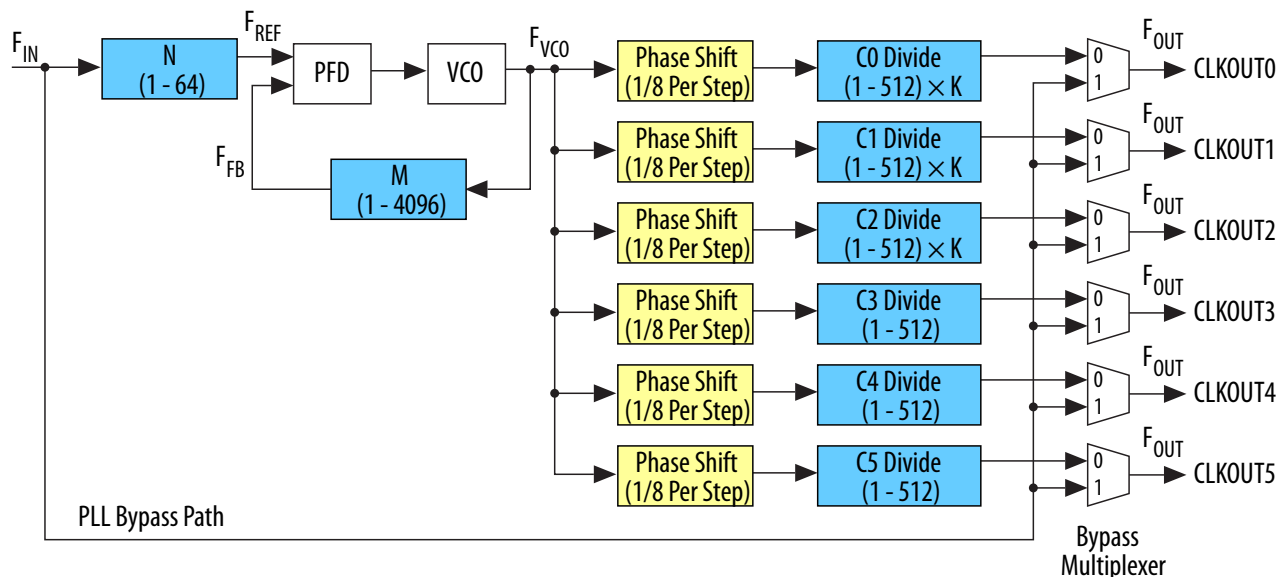
The SDRAM PLL has the following additional feature:

- Phase shift of 1/8 per step
 - Phase shift range is 0 to 7

FREF, FVCO, and FOUT Equations

Figure 3-2: PLL Block Diagram

Values listed for M, N, and C are actually one greater than the values stored in the CSRs.



$$F_{REF} = F_{IN} / N$$

$$FVCO = F_{REF} \times M = F_{IN} \times M/N$$

$$FOUT = F_{VCO} / (C_i \times K) = F_{REF} \times M / (C_i \times K) = (F_{IN} \times M) / (N \times C_i \times K)$$

Table 3-2: FREF, FVCO, and FOUT Equation variables

Variable	Value	Description
FVC	= VCO frequency	-
FIN	= Input frequency	-
FREF	= Reference frequency	-
C _i	= Post-scale counter	i is 0-5 for each of the six counters
K	= Internal post-scale counter in main PLL	reset values are K = 2 for C0 K=4 for C1 and C2
M	= numer + 1	Part of clock feedback path. VCO register is used to program M value. (Range 1 to 4096)
N	= denom + 1	Part of input clock path. VCO register is used to program N value. (Range 0 to 64)

Note: The reset values of numer and denom are 1 so that at reset, the M value is 2 and the N value is 2.

The vco register is used to program the M and N values. In the table below you can see which sections of the vco bit field are used to set the values of M and N.

Table 3-3: VCO Register

Name	Bit	Reset	Range	Description
numer	3:15	0x1	0 to 4095	Numerator in VCO output frequency equation. Note: Bit 15 reserved.
denom	16:21	0x1	0 to 63	Denominator in VCO output frequency equation.

Unused clock outputs should be set to a safe frequency such as 50 MHz to reduce power consumption and improve system stability.

Related Information

[Clock Manager Address Map and Register Definitions](#) on page 3-23

For the full bit field of the vco register, refer to the Address Map and Register Definitions section.

Dividers

Dividers subdivide the C0-C15 clocks produced by the PLL to lower frequencies. The main PLL C0-C2 clocks have an additional internal post-scale counter.

Clock Gating

Clock gating enables and disables clock signals. Refer to the Peripheral PLL Group Enable Register (en) for more information on what clocks can be gated.

Related Information

[Clock Manager Address Map and Register Definitions](#) on page 3-23

Control and Status Registers

The *Clock Manager* contains registers used to configure and observe the clock manager.

Hardware-Managed and Software-Managed Clocks

When changing values on clocks, the terms hardware-managed and software-managed define who is responsible for successful transitions. Software-managed clocks require that software manually gate any clock affected by the change, wait for any PLL lock if required, then ungate the clocks. Hardware-managed clocks use hardware to ensure that a glitch-free transition to a new clock value occurs. There are three hardware-managed sets of clocks in the HPS, namely, clocks generated from the main PLL outputs C0, C1, and C2. All other clocks in the HPS are software-managed clocks.

Clock Groups

The clock manager contains one clock group for each PLL and one clock group for the HPS_CLK1 pin.

HPS_CLK1 and HPS_CLK2 are powered by the HPS reset and clock input pins power supply (V_{CCRSTCLK_HPS}). For more information on V_{CCRSTCLK_HPS} refer to the Arria V Device Datasheet.

OSC1 Clock Group

The clock in the OSC1 clock group is derived directly from the HPS_CLK1 pin. This clock is never gated or divided.

HPS_clk1 is used as a PLL input and also by HPS logic that does not operate on a clock output from a PLL.

Table 3-4: OSC1 Clock Group Clock

Name	Frequency	Clock Source	Destination
osc1_clk	0 to 100 MHz	HPS_CLK1 pin	OSC1-driven peripherals. Refer to "Main Clock Group Clocks".

Main Clock Group

The main clock group consists of a PLL, dividers, and clock gating. The clocks in the main clock group are derived from the main PLL. The main PLL is always sourced from the HPS_CLK1 pin of the device.

Table 3-5: Main PLL Output Assignments

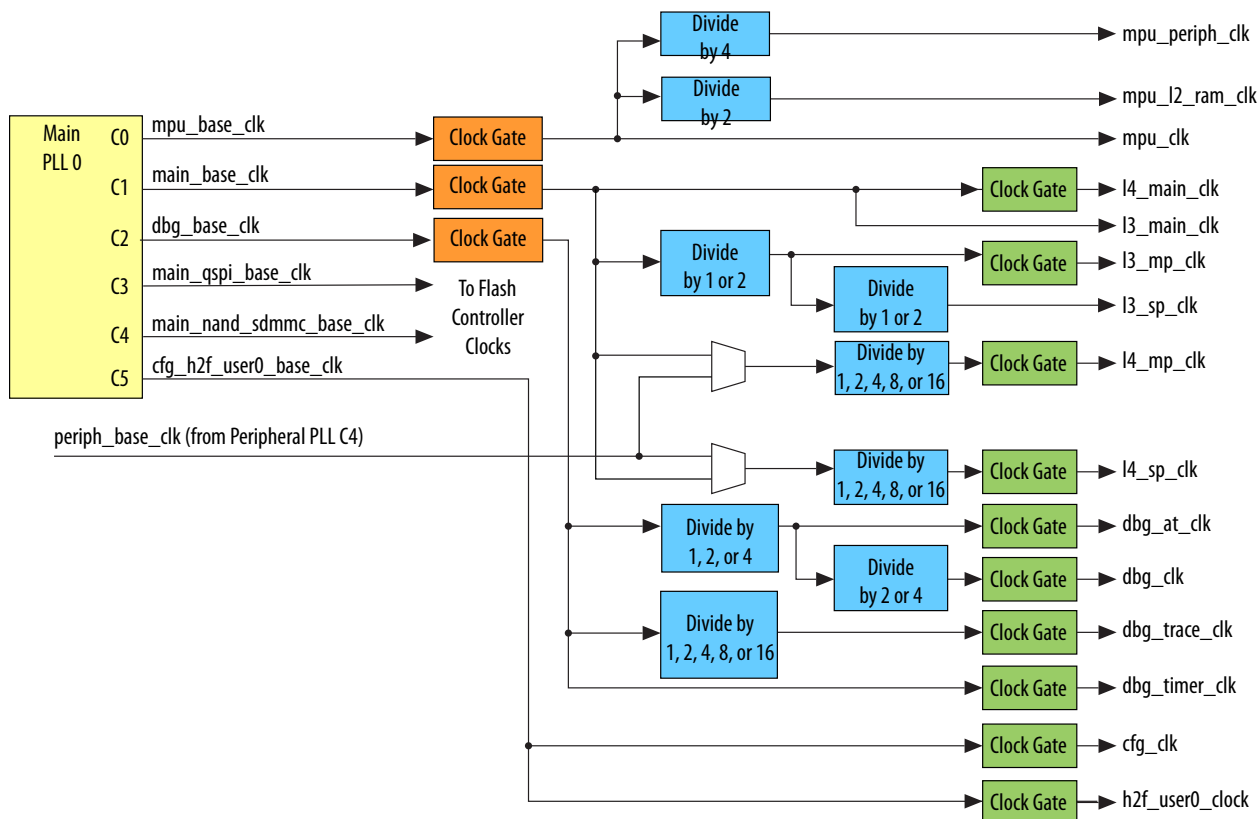
PLL	Output Counter	Clock Name	Frequency	Phase Shift Control
Main	C0	mpu_base_clk	osc1_clk to varies (7)	No
	C1	main_base_clk	osc1_clk to varies (7)	No
	C2	dbg_base_clk	osc1_clk/4 to mpu_base_clk/2	No
	C3	main_qspi_base_clk	Up to 432 MHz	No
	C4	main_nand_sdmmc_base_clk	Up to 250 MHz for the NAND flash controller and up to 200 MHz for the SD/MMC controller	No
	C5	cfg_h2f_user0_base_clk	osc1_clk to 125 MHz for driving configuration and 100 MHz for the user clock	No

The counter outputs from the main PLL can have their frequency further divided by programmable dividers external to the PLL. Transitions to a different divide value occur on the fastest output clock, one clock cycle prior to the slowest clock's rising edge. For example the clock transitions on cycle 15 of the divide-by-16 divider for the main C2 output and cycle 3 of the divide-by-4 divider for the main C0 output.

The following figure shows how each counter output from the main PLL can have its frequency further divided by programmable post-PLL dividers. Green-colored clock gating logic is directly controlled by software writing to a register. Orange-colored clock gating logic is controlled by hardware. Orange-colored clock gating logic allows hardware to seamlessly transition a synchronous set of clocks, for example, all the MPU subsystem clocks.

⁽⁷⁾ The maximum frequency depends on the speed grade of the device.

Figure 3-3: Main Clock Group Divide and Gating



The clocks derived from main PLL C0-C2 outputs are hardware-managed, meaning hardware ensures that a clean transition occurs, and can have the following control values changed dynamically by software write accesses to the control registers:

- PLL bypass
- PLL numerator, denominator, and counters
- External dividers

For these registers, hardware detects that the write has occurred and performs the correct sequence to ensure that a glitch-free transition to the new clock value occurs. These clocks can pause during the transition.

Table 3-6: Main Clock Group Clocks

System Clock Name	Frequency	Constraints and Notes
mpu_clk	Main PLL C0	Clock for MPU subsystem, including CPU0 and CPU1
mpu_l2_ram_clk	mpu_clk/2	Clock for MPU level 2 (L2) RAM

System Clock Name	Frequency	Constraints and Notes
mpu_periph_clk	mpu_clk/4	Clock for MPU snoop control unit (SCU) peripherals, such as the general interrupt controller (GIC)
l3_main_clk	Main PLL C1	Clock for L3 main switch
l3_mp_clk	l3_main_clk/2	Clock for L3 master peripherals (MP) switch
l3_sp_clk	l3_mp_clk or l3_mp_clk/2	Clock for L3 slave peripherals (SP) switch
l4_main_clk	Main PLL C1	Clock for L4 main bus
l4_mp_clk	osc1_clk/16 to 100 MHz divided from main PLL C1 or peripheral PLL C4	Clock for L4 MP bus
l4_sp_clk	osc1_clk/16 to 100 MHz divided from main PLL C1 or peripheral PLL C4	Clock for L4 SP bus
dbg_at_clk	osc1_clk/4 to main PLL C2/2	Clock for CoreSight™ debug trace bus
dbg_trace_clk	osc1_clk/16 to main PLL C2	Clock for CoreSight™ debug Trace Port Interface Unit (TPIU)
dbg_timer_clk	osc1_clk to main PLL C2	Clock for the trace timestamp generator
dbg_clk (8)	dbg_at_clk/2 or dbg_at_clk/4	Clock for Debug Access Port (DAP) and debug peripheral bus
main_qspi_clk	Main PLL C3	Quad SPI flash internal logic clock
main_nand_sdmmc_clk	Main PLL C4	Input clock to flash controller clocks block
cfg_clk	osc1_clk to 100_MHz divided from main PLL C5	FPGA manager configuration clock

(8) dbg_clk must be at least twice as fast as the JTAG clock.

System Clock Name	Frequency	Constraints and Notes
h2f_user0_clock	osc1_clk to 100_MHz divided from main PLL C5	Auxiliary user clock to the FPGA fabric

Changing Values That Affect Main Clock Group PLL Lock

To change any value that affects the VCO lock of the main clock group PLL including the hardware-managed clocks, software must put the main PLL in bypass mode, which causes all the main PLL output clocks to be driven by the osc1_clk clock. Software must detect PLL lock by reading the lock status register prior to taking the main PLL out of bypass mode.

Once a PLL is locked, changes to any PLL VCO frequency that are 20 percent or less do not cause the PLL to lose lock. Iteratively changing the VCO frequency in increments of 20 percent or less allow a slow ramp of the VCO base frequency without loss of lock. For example, to change a VCO frequency by 40% without losing lock, change the frequency by 20%, then change it again by 16.7%.

Peripheral Clock Group

The peripheral clock group consists of a PLL, dividers, and clock gating. The clocks in the peripheral clock group are derived from the peripheral PLL. The peripheral PLL can be programmed to be sourced from the HPS_CLK1 pin, the HPS_CLK2 pin, or the f2h_periph_ref_clk clock provided by the FPGA fabric.

The FPGA fabric must be configured with an image that provides the f2h_periph_ref_clk before selecting it as the clock source. If the FPGA must be reconfigured and f2h_periph_ref_clk is being used by modules in the HPS, an alternate clock source must be selected prior to reconfiguring the FPGA.

Clocks that always use the peripheral PLL output clocks as the clocks source are:

- emac0_clk
- emac1_clk
- usb_mp_clk
- spi_m_clk
- gpio_db_clk
- h2f_user1_clk

In addition, clocks that may use the peripheral PLL output clocks as the clock source are:

- sdmmc_clk
- nand_clk
- qspi_clk
- l4_mp_clk
- l4_sp_clk

The counter outputs from the main PLL can have their frequency further divided by external dividers. Transitions to a different divide value occur on the fastest output clock, one clock cycle prior to the slowest clock's rising edge. For example, the clock transitions on cycle 15 of the divide-by-16 divider for the main C2 output and cycle 3 of the divide-by-4 divider for the C1 output.

Table 3-7: Peripheral PLL Output Assignments

PLL	Output Counter	Clock Name	Frequency	Phase Shift Control
Peripheral	C0	emac0_base_clk	Up to 250 MHz	No
	C1	emac1_base_clk	Up to 250 MHz	No
	C2	periph_qspi_base_clk	Up to 432 MHz	No
	C3	periph_nand_sdmmc_base_clk	Up to 250 MHz for the NAND flash controller and up to 200 MHz for the SD/MMC controller	No
	C4	periph_base_base_clk	Up to 240 MHz for the SPI masters and up to 200 MHz for the scan manager	No
	C5	h2f_user1_base_clk	osc1_clk to 100 MHz	No

The following figure shows programmable post-PLL dividers and clock gating for the peripheral clock group. Clock gate blocks in the diagram indicate clocks that may be gated off under software control. Software is expected to gate these clocks off prior to changing any PLL or divider settings that might create incorrect behavior on these clocks.

Figure 3-4: Peripheral Clock Group Divide and Gating

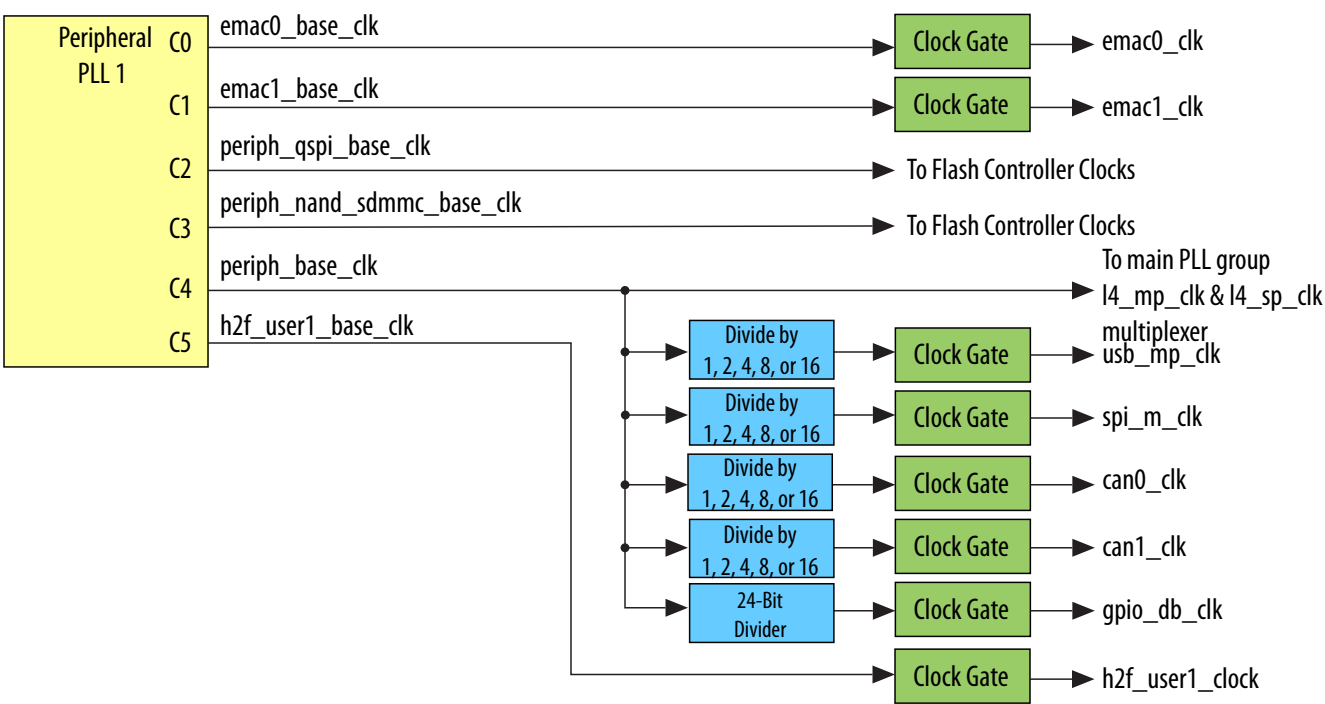


Table 3-8: Peripheral Clock Group Clocks

System Clock Name	Frequency	Divided From	Constraints and Notes
usb_mp_clk	Up to 200 MHz	Peripheral PLL C4	Clock for USB
spi_m_clk	Up to 240 MHz for the SPI masters and up to 200 MHz for the scan manager	Peripheral PLL C4	Clock for L4 SPI master bus and scan manager
emac0_clk	Up to 250 MHz	Peripheral PLL C0	EMAC0 clock. The 250 MHz clock is divided internally by the EMAC into the typical 125/25/2.5 MHz speeds for 1000/100/10 Mbps operation.

System Clock Name	Frequency	Divided From	Constraints and Notes
emac1_clk	Up to 250 MHz	Peripheral PLL C1	EMAC1 clock The 250 MHz clock is divided internally by the EMAC into the typical 125/25/2.5 MHz speeds for 1000/100/10 Mbps operation.
l4_mp_clk	Up to 100 MHz	Main PLL C1 or peripheral PLL C4	Clock for L4 master peripheral bus
l4_sp_clk	Up to 100 MHz	Main PLL C1 or peripheral PLL C4	Clock for L4 slave peripheral bus
gpio_db_clk	Up to 1 MHz	Peripheral PLL C4	Used to debounce GPIO0, GPIO1, and GPIO2
h2f_user1_clock	Peripheral PLL C5	Peripheral PLL C5	Auxiliary user clock to the FPGA fabric

Flash Controller Clocks

Flash memory peripherals can be driven by the main PLL, the peripheral PLL, or from clocks provided by the FPGA fabric.

Figure 3-5: Flash Peripheral Clock Divide and Gating

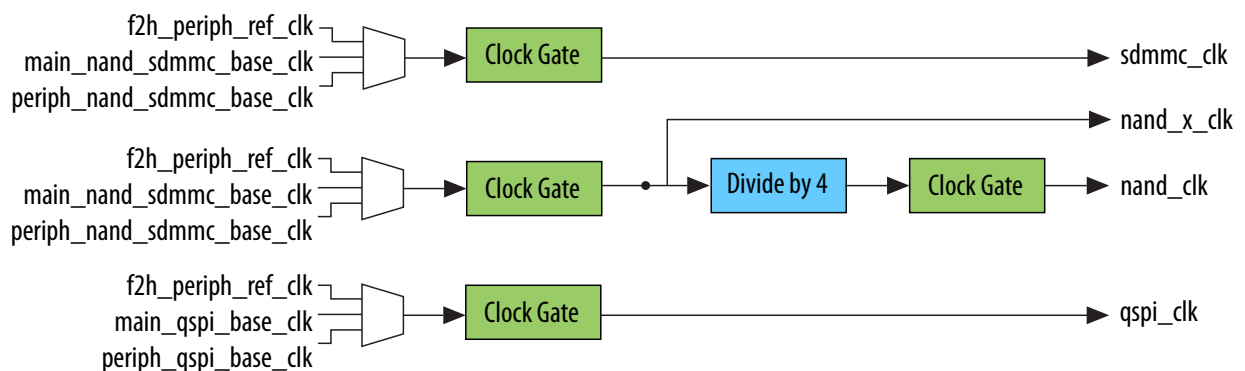


Table 3-9: Flash Controller Clocks

System Clock Name	Frequency	Divided From	Constraints and Notes
qspi_clk	Up to 432 MHz	Peripheral PLL C2, main PLL C3, or f2h_periph_ref_clk	Clock for quad SPI, typically 108 and 80 MHz
nand_x_clk	Up to 250 MHz	Peripheral PLL C3, main PLL C4, or f2h_periph_ref_clk	NAND flash controller master and slave clock
nand_clk	nand_x_clk/4	Peripheral PLL C3, main PLL C4, or f2h_periph_ref_clk	Main clock for NAND flash controller, sets base frequency for NAND transactions
sddmmc_clk	Up to 200 MHz	Peripheral PLL C3, main PLL C4, or f2h_periph_ref_clk	• Less than or equal to memory maximum operating frequency • 45% to 55% duty cycle • Typical frequencies are 25 and 50 MHz • SD/MMC has a subclock tree divided down from this clock

SDRAM Clock Group

The SDRAM clock group consists of a PLL and clock gating. The clocks in the SDRAM clock group are derived from the SDRAM PLL. The SDRAM PLL can be programmed to be sourced from the HPS_CLK1 pin, the HPS_CLK2 pin, or the f2h_sdram_ref_clk clock provided by the FPGA fabric.

The FPGA fabric must be configured with an image that provides the f2h_sdram_ref_clk before selecting it as the clock source. If the FPGA must be reconfigured and the f2h_sdram_ref_clk is the clock source for the SDRAM clock group, an alternate clock source must be selected prior to reconfiguring the FPGA.

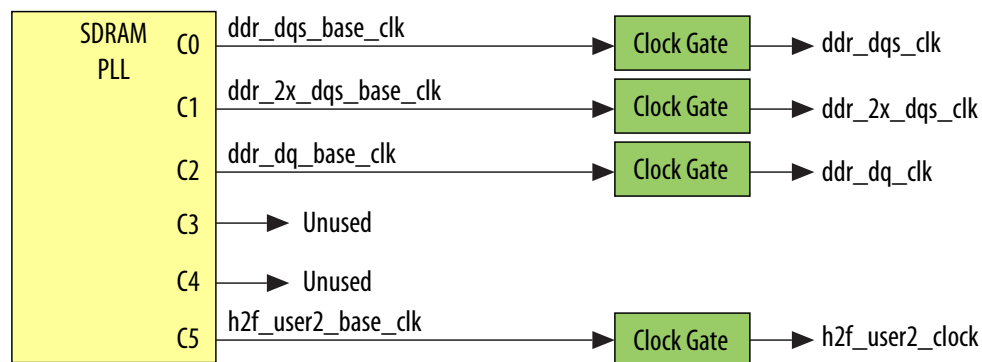
The counter outputs from the SDRAM PLL can be gated off directly under software control. The divider values for each clock are set by registers in the clock manager.

Table 3-10: SDRAM PLL Output Assignments

PLL	Output Counter	Clock Name	Frequency	Phase Shift Control
SDRAM	C0	ddr_dqs_base_clk	Varies ⁽⁹⁾	Yes
	C1	ddr_2x_dqs_base_clk	ddr_dqs_base_clk x 2	Yes
	C2	ddr_dq_base_clk	ddr_dqs_base_clk	Yes
	C5	h2f_user2_base_clk	osc1_clk to varies ⁽⁹⁾	Yes

The following figure shows clock gating for SDRAM PLL clock group. Clock gate blocks in the diagram indicate clocks which may be gated off under software control. Software is expected to gate these clocks off prior to changing any PLL or divider settings that might create incorrect behavior on these clocks.

Figure 3-6: SDRAM Clock Group Divide and Gating



The SDRAM PLL output clocks can be phase shifted in real time in increments of 1/8 the VCO frequency. Maximum number of phase shift increments is 4096.

Table 3-11: SDRAM Clock Group Clocks

Name	Frequency	Constraints and Notes
ddr_dqs_clk	SDRAM PLL C0	Clock for MPFE, single-port controller, CSR access, and PHY

⁽⁹⁾ The maximum frequency depends on the speed grade of the device.

Name	Frequency	Constraints and Notes
ddr_2x_dqs_clk	SDRAM PLL C1	Clock for PHY
ddr_dq_clk	SDRAM PLL C2	Clock for PHY
h2f_user2_clock	SDRAM PLL C5	Auxiliary user clock to the FPGA fabric

Resets

Cold Reset

Cold reset places the hardware-managed clocks into safe mode, the software-managed clocks into their default state, and asynchronously resets all registers in the clock manager.

Related Information

[Safe Mode](#) on page 3-17

Warm Reset

Registers in the clock manager control how the clock manager responds to warm reset. Typically, software places the clock manager into a safe state in order to generate a known set of clocks for the ROM code to boot the system. The behavior of the system on warm reset as a whole, including how the FPGA fabric, boot code, and debug systems are configured to behave, must be carefully considered when choosing how the clock manager responds to warm reset.

The reset manager can request that the clock manager go into safe mode as part of the reset manager's warm reset sequence. Before asserting safe mode to the clock manager, the reset manager ensures that the reset signal is asserted to all modules that receive warm reset.

Related Information

[Reset Manager](#) on page 4-1

For more information, refer to "Reset Sequencing" in the *Reset Manager* chapter.

Safe Mode

Safe mode is enabled in the HPS by a cold reset. Also, if the `ensfmdwr` bit in the `ctrl` register is set, clock manager responds to the Safe Mode request from Reset Manager on a warm reset and sets the Safe Mode bit. No other control register bits are affected by the safe mode request from the Reset Manager.

Note: By default the preloader generated by the SoC EDS tool sets the `ensfmdwr` bit after bringing the clocks out from reset. In order to ensure the stability of the system, Intel recommends that you do not clear the `ensfmdwr` bit during any of the later HPS boot stages.

When safe mode is enabled, the main PLL hardware-managed clocks (C0-C2) are bypassed to `osc1_clk` clock and are directly generated from `osc1_clk`. While in safe mode, clock manager register settings, which control clock behavior, are not changed. However, the hardware bypasses these settings and uses safe, default settings.

The hardware-managed clocks are forced to their safe mode values such that the following conditions occur:

- The hardware-managed clocks are bypassed to `osc1_clk`, including counters in the main PLL.
- Programmable dividers select the reset default values.
- The flash controller clocks multiplexer selects the output from the peripheral PLL.
- All clocks are enabled.

A write by software is the only way to clear the safe mode bit (`safemode`) of the `ctrl` register.

Note: Before coming out of safe mode, all registers and clocks must be configured correctly. It is possible to program the clock manager in such a way that only a cold reset can return the clocks to a functioning state. Intel strongly recommends using provided libraries to configure and control HPS clocks.

Interrupts

The clock manager provides one interrupt output which is enabled using the interrupt enable register (`intren`). The interrupt is the OR of the bits in the interrupt status register (`inter`) that indicate lock and loss of lock for each of the three PLLs.

Clock Usage By Module

The following table lists every clock input generated by the clock manager to all modules in the HPS. System clock names are global for the entire HPS and system clocks with the same name are phase-aligned at all endpoints.

Table 3-12: Clock Usage By Module

Module Name	System Clock Name	Use
MPU subsystem	<code>mpu_clk</code>	Main clock for the MPU subsystem
	<code>mpu_periph_clk</code>	Peripherals inside the MPU subsystem
	<code>dbg_at_clk</code>	Trace bus
	<code>dbg_clk</code>	Debug
	<code>mpu_l2_ram_clk</code>	L2 cache and Accelerator Coherency Port (ACP) ID mapper
	<code>l4_mp_clk</code>	ACP ID mapper control slave

Module Name	System Clock Name	Use
Interconnect	l3_main_clk	L3 main switch
	dbg_at_clk	System Trace Macrocell (STM) slave and Embedded Trace Router (ETR) master connections
	dbg_clk	DAP master connection
	l3_mp_clk	L3 master peripheral switch
	l4_mp_clk	L4 MP bus, Secure Digital (SD) / MultiMediaCard (MMC) master, and EMAC masters
	usb_mp_clk	USB masters and slaves
	nand_x_clk	NAND master
	cfg_clk	FPGA manager configuration data slave
	l3_sp_clk	L3 slave peripheral switch
	l3_main_clk	L4 SPIS bus master
	mpu_l2_ram_clk	ACP ID mapper slave and L2 master connections
	osc1_clk	L4 OSC1 bus master
	spi_m_clk	L4 SPIM bus master
	l4_sp_clk	L4 SP bus master
	l4_mp_clk	Quad SPI bus slave
Boot ROM	l3_main_clk	Boot ROM
On-chip RAM	l3_main_clk	On-chip RAM



Module Name	System Clock Name	Use
DMA controller	l4_main_clk	DMA
	dbg_at_clk	Synchronous to the STM module
	l4_mp_clk	Synchronous to the quad SPI flash
FPGA manager	cfg_clk	Control block (CB) data interface and configuration data slave
	l4_mp_clk	Control slave
HPS-to-FPGA bridge	l3_main_clk	Data slave
	l4_mp_clk	Global programmer's view (GPV) slave
FPGA-to-HPS bridge	l3_main_clk	Data master
	l4_mp_clk	GPV slave
Lightweight HPS-to-FPGA bridge	l4_mp_clk	GPV masters and the data and GPV slave
Quad SPI flash controller	l4_mp_clk	Control slave
	qspi_clk	Reference for serialization
SD/MMC controller	l4_mp_clk	Master and slave
	sdmmc_clk	SD/MMC internal logic
EMAC 0	l4_mp_clk	Master
	emac0_clk	EMAC 0 internal logic
	osc1_clk	IEEE 1588 timestamp

Module Name	System Clock Name	Use
EMAC 1	l4_mp_clk	Master
	emac1_clk	EMAC 1 internal logic
	osc1_clk	IEEE 1588 timestamp
USB 0	usb_mp_clk	Master and Slave
USB 1	usb_mp_clk	Master and Slave
NAND flash controller	nand_x_clk	NAND high-speed master and slave
	nand_clk	NAND flash
OSC1 timer 0	osc1_clk	OSC1 timer 0
OSC1 timer 1	osc1_clk	OSC1 timer 1
SP timer 0	l4_sp_clk	SP timer 0
SP timer 1	l4_sp_clk	SP timer 1
I ² C controller 0	l4_sp_clk	I ² C 0
I ² C controller 1	l4_sp_clk	I ² C 1
I ² C controller 2	l4_sp_clk	I ² C 2
I ² C controller 3	l4_sp_clk	I ² C 3
UART controller 0	l4_sp_clk	UART 0
UART controller 1	l4_sp_clk	UART 1

Module Name	System Clock Name	Use
GPIO interface 0	l4_mp_clk	Slave
	gpio_db_clk	Debounce
GPIO interface 1	l4_mp_clk	Slave
	gpio_db_clk	Debounce
GPIO interface 2	l4_mp_clk	Slave
	gpio_db_clk	Debounce
System manager	osc1_clk	System manager
SDRAM subsystem	l4_sp_clk	Control slave
	ddr_dq_clk	Off-chip data
	ddr_dqs_clk	MPFE, single-port controller, CSRs, and PHY
	ddr_2x_dqs_clk	Off-chip strobe data
	mpu_l2_ram_clk	Slave connected to MPU subsystem L2 cache
	l3_main_clk	Slave connected to L3 interconnect
L4 watchdog timer 0	osc1_clk	L4 watchdog timer 0
L4 watchdog timer 1	osc1_clk	L4 watchdog timer 1
SPI master controller 0	spi_m_clk	SPI master 0
SPI master controller 1	spi_m_clk	SPI master 1
SPI slave controller 0	l4_main_clk	SPI slave 0

Module Name	System Clock Name	Use
SPI slave controller 1	l4_main_clk	SPI slave 1
Debug subsystem	l4_mp_clk	System bus
	dbg_clk	Debug
	dbg_at_clk	Trace bus
	dbg_trace_clk	Trace port
Reset manager	osc1_clk	Reset manager
	l4_sp_clk	Slave
Scan manager	spl_m_clk	Scan manager
Timestamp generator	dbg_timer_clk	Timestamp generator

Clock Manager Address Map and Register Definitions

The address map and register definitions for the HPS-FPGA bridge consist of the following regions:

- Clock Manager Module

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter.
- <https://www.intel.com/content/www/us/en/programmable/hps/arria-v/hps.html>

2021.05.07

av_5v4



Subscribe



Send Feedback

The reset manager generates module reset signals based on reset requests from the various sources in the HPS and FPGA fabric, and software writing to the module-reset control registers. The reset manager ensures that a reset request from the FPGA fabric can occur only after the FPGA portion of the system-on-a-chip (SoC) device is configured.

The HPS contains multiple reset domains. Each reset domain can be reset independently. A reset may be initiated externally, internally or through software.

Table 4-1: HPS Reset Domains

Domain Name	Domain Logic
TAP	JTAG test access port (TAP) controller, which is used by the debug access port (DAP).
Debug	All debug logic including most of the DAP, CoreSight™ components connected to the debug peripheral bus, trace, the microprocessor unit (MPU) subsystem, and the FPGA fabric.
System	All HPS logic except what is in the TAP and debug reset domains. Includes non-debug logic in the FPGA fabric connected to the HPS reset signals.

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

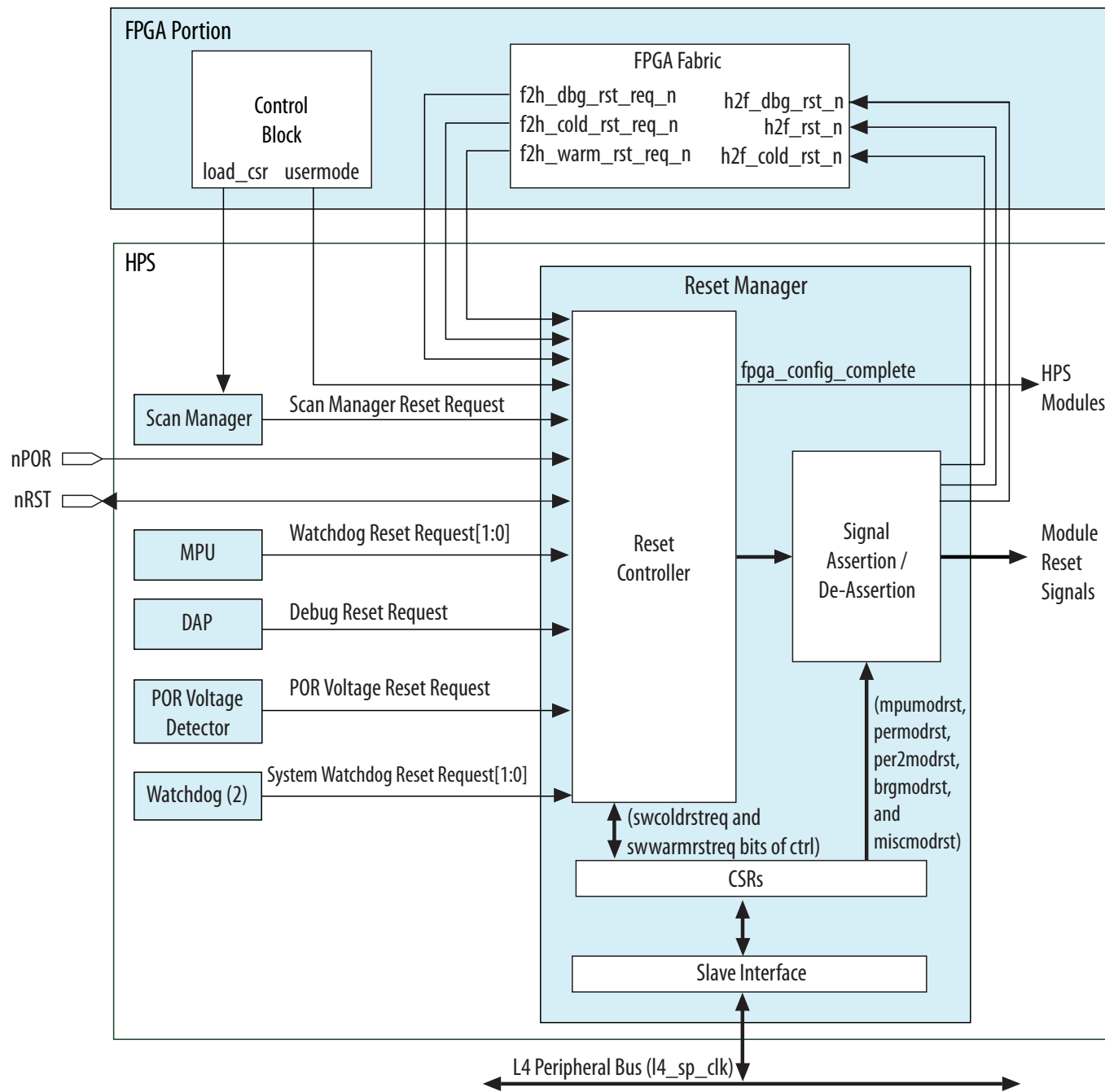
The HPS supports the following reset types:

- System cold reset
 - Used to ensure the HPS is placed in a default state sufficient for software to boot
 - Triggered by a power-on reset and other sources
 - Resets all HPS logic that can be reset
 - Affects all reset domains
- System warm reset
 - Occurs after HPS has already completed a cold reset
 - Used to recover system from a non-responsive condition
 - Resets a subset of the HPS state reset by a cold reset
 - Only affects the system reset domain, which allows debugging (including trace) to operate through the warm reset
- Debug reset
 - Used to recover debug logic from a non-responsive condition
 - Only affects the debug reset domain

Reset Manager Block Diagram and System Integration

The following figure shows a block diagram of the reset manager in the SoC device. For clarity, reset-related handshaking signals to other HPS modules and to the clock manager module are omitted.

Figure 4-1: Reset Manager Block Diagram



HPS External Reset Sources

The following table lists the reset sources external to the HPS. All signals are synchronous to the `osc1_clk` clock. The reset signals from the HPS to the FPGA fabric must be synchronized to your user logic clock domain.

Table 4-2: HPS External Reset Sources

Source	Description
f2h_cold_rst_req_n	Cold reset request from FPGA fabric (active low)
f2h_warm_rst_req_n	Warm reset request from FPGA fabric (active low)
f2h_dbg_rst_req_n	Debug reset request from FPGA fabric (active low)
h2f_cold_rst_n	Cold-only reset to FPGA fabric (active low)
h2f_rst_n	Cold or warm reset to FPGA fabric (active low)
h2f_dbg_rst_n	Debug reset (dbg_rst_n) to FPGA fabric (active low)
load_csr	Cold-only reset from FPGA control block (CB) and scan manager
nPOR	Power-on reset pin (active low)
nRST	Warm reset pin (active low)

nRST and nPOR are powered by the HPS reset and clock input pins power supply ($V_{CCRSTCLK\_HPS}$). For more information on $V_{CCRSTCLK\_HPS}$, refer to the *Arria V Device Datasheet*.

Related Information

[Arria V GX, GT, SX, and ST Device Datasheet](#)

For information about the required duration of reset request signal assertion, refer to the *Arria V Device Datasheet*.

Reset Controller

The reset controller performs the following functions:

- Accepts reset requests from the FPGA CB, FPGA fabric, modules in the HPS, and reset pins
- Generates an individual reset signal for each module instance for all modules in the HPS
- Provides reset handshaking signals to support system reset behavior

The reset controller generates module reset signals from external reset requests and internal reset requests. External reset requests originate from sources external to the reset manager. Internal reset requests originate from control registers in the reset manager.

The reset controller supports the following cold reset requests:

- Power-on reset (POR) voltage monitor
- Cold reset request pin (nPOR)
- FPGA fabric
- FPGA CB and scan manager
- Software cold reset request bit (swcoldrstreq) of the control register (ctrl)

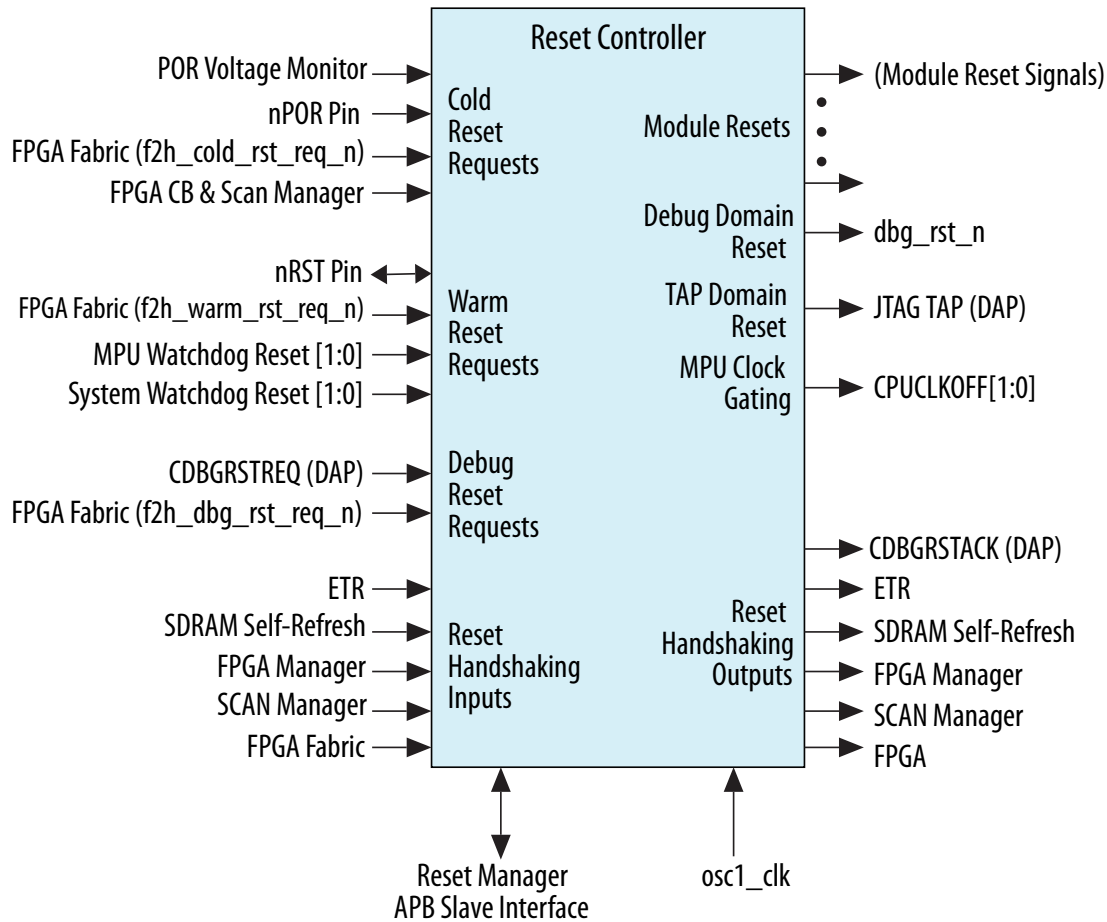
The reset controller supports the following warm reset requests:

- Warm reset request pin (nRST)
- FPGA fabric
- Software warm reset request bit (swarmrstreq) of the ctrl register
- MPU watchdog reset requests for CPU0 and CPU1
- System watchdog timer 0 and 1 reset requests

The reset controller supports the following debug reset requests:

- CDBGIRSTREQ from DAP
- FPGA fabric

Figure 4-2: Reset Controller Signals



Module Reset Signals

The following tables list the module reset signals. The module reset signals are organized in groups for the MPU, peripherals, bridges.

In the following tables, columns marked for Cold Reset, Warm Reset, and Debug Reset denote reset signals asserted by each type of reset. For example, writing a 1 to the swarmrstreq bit in the ctrl register resets all the modules that have a checkmark in the Warm Reset column.

The column marked for Software Deassert denotes reset signals that are left asserted by the reset manager.

Table 4-3: MPU Group, Generated Module Resets

Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
mpu_cpu_rst_n[0]	Resets each processor in the MPU	System	X	X		
mpu_cpu_rst_n[1]	Resets each processor in the MPU	System	X	X		X
mpu_wd_rst_n	Resets both per-processor watchdogs in the MPU	System	X	X		
mpu_scu_periph_rst_n	Resets Snoop Control Unit (SCU) and peripherals	System	X	X		
mpu_l2_rst_n	Level 2 (L2) cache reset	System	X	X		

Table 4-4: PER Group, Generated Module Resets

Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
emac_rst_n[1:0]	Resets each EMAC	System	X	X		X
usb_rst_n[1:0]	Resets each USB	System	X	X		X
nand_flash_rst_n	Resets NAND flash controller	System	X	X		X
qspi_flash_rst_n	Resets quad SPI flash controller	System	X	X		X
watchdog_rst_n[1:0]	Resets each system watchdog timer	System	X	X		X
osc1_timer_rst_n[1:0]	Resets each OSC1 timer	System	X	X		X
sp_timer_rst_n[1:0]	Resets each SP timer	System	X	X		X

Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
i2c_rst_n[3:0]	Resets each I ² C controller	System	X	X		X
uart_rst_n[1:0]	Resets each UART	System	X	X		X
spim_rst_n[1:0]	Resets SPI master controller	System	X	X		X
spis_rst_n[1:0]	Resets SPI slave controller	System	X	X		X
sdmmc_rst_n	Resets SD/MMC controller	System	X	X		X
gpio_rst_n[2:0]	Resets each GPIO interface	System	X	X		X
dma_rst_n	Resets DMA controller	System	X	X		X
sdram_rst_n	Resets SDRAM subsystem (resets logic associated with cold or warm reset)	System	X	X		X

Table 4-5: PER2 Group, Generated Module Resets

Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
dma_periph_if_rst_n[7:0]	DMA controller request interface from FPGA fabric to DMA controller	System	X	X		X

Table 4-6: Bridge Group, Generated Module Resets

Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
hps2fpga_bridge_rst_n	Resets HPS-to-FPGA AMBA* Advanced eXtensible Interface (AXI) bridge	System	X	X		X
fpga2hps_bridge_rst_n	Resets FPGA-to-HPS AXI bridge	System	X	X		X
lwhps2fpga_bridge_rst_n	Resets lightweight HPS-to-FPGA AXI bridge	System	X	X		X

Table 4-7: MISC Group, Generated Module Resets

Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
boot_rom_rst_n	Resets boot ROM	System	X	X		
onchip_ram_rst_n	Resets on-chip RAM	System	X	X		
sys_manager_rst_n	Resets system manager (resets logic associated with cold or warm reset)	System	X	X		
sys_manager_cold_rst_n	Resets system manager (resets logic associated with cold reset only)	System	X			
fpga_manager_rst_n	Resets FPGA manager	System	X	X		
acp_id_mapper_rst_n	Resets ACP ID mapper	System	X	X		
h2f_rst_n	Resets user logic in FPGA fabric (resets logic associated with cold or warm reset)	System	X	X		
h2f_cold_rst_n	Resets user logic in FPGA fabric (resets logic associated with cold reset only)	System	X			
rst_pin_rst_n	Pulls nRST pin low	System		X		
timestamp_cold_rst_n	Resets debug timestamp to 0x0	System	X			
clk_manager_cold_rst_n	Resets clock manager (resets logic associated with cold reset only)	System	X			
scan_manager_rst_n	Resets scan manager	System	X	X		
frz_ctrl_cold_rst_n	Resets freeze controller (resets logic associated with cold reset only)	System	X			
sys_dbg_rst_n	Resets debug masters and slaves connected to L3 interconnect and level 4 (L4) buses	System	X	X		



Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
dbg_rst_n	Resets debug components including DAP, trace, MPU debug logic, and any user debug logic in the FPGA fabric	Debug	X		X	
tap_cold_rst_n	Resets portion of TAP controller in the DAP that must be reset on a cold reset	TAP	X			
sdram_cold_rst_n	Resets SDRAM subsystem (resets logic associated with cold reset only)	System	X			

Table 4-8: L3 Group, Generated Module Resets

Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
l3_rst_n	Resets L3 interconnect and L4 buses	System	X	X		

Modules Requiring Software Deassert

The reset manager leaves the reset signal asserted on certain modules even after the rest of the HPS has come out of reset. These modules are likely to require software configuration before they can safely be taken out of reset.

When a module that has been held in reset is ready to start running, software can deassert the reset signal by writing to the appropriate register, shown in the following table.

Table 4-9: Module Reset Signals and Registers

HPS Peripheral	Reset Register	Module Reset Signal	Reset Group
MPU	mpumodrst	mpu_cpu_rst_n[1]	MPU
Ethernet MAC	permodrst	emac_rst_n[1:0]	PER
USB 2.0 OTG	permodrst	usb_rst_n[1:0]	PER
NAND	permodrst	nand_flash_rst_n	PER
Quad SPI	permodrst	qspi_flash_rst_n	PER
Watchdog	permodrst	watchdog_rst_n[1:0]	PER
Timer	permodrst	osc1_timer_rst_n[1:0]	PER
Timer	permodrst	sp_timer_rst_n[1:0]	PER
I ² C	permodrst	i2c_rst_n[3:0]	PER

HPS Peripheral	Reset Register	Module Reset Signal	Reset Group
UART	permodrst	uart_rst_n[1:0]	PER
SPI Master	permodrst	spim_rst_n[1:0]	PER
SPI Slave	permodrst	spis_rst_n[1:0]	PER
SD/MMC	permodrst	sdmmc_rst_n	PER
GPIO	permodrst	gpio_rst_n[2:0]	PER
DMA	permodrst	dma_rst_n	PER
SDRAM	permodrst	sdram_rst_n	PER
DMA Peripheral Interfaces	per2modrst	dma_periph_if_rst_n[7:0]	PER2
HPS-to-FPGA Bridge	brgmodrst	hps2fpga_bridge_rst_n	Bridge
FPGA-to-HPS Bridge	brgmodrst	fpga2hps_bridge_rst_n	Bridge
Lightweight HPS-to-FPGA Bridge	brgmodrst	lw_hps2fpga_bridge_rst_n	Bridge

Slave Interface and Status Register

The reset manager slave interface is used to control and monitor the reset states.

The status register (stat) in the reset manager contains the status of the reset requester. The register contains a bit for each monitored reset request. The stat register captures all reset requests that have occurred. Software is responsible for clearing the bits.

During the boot process, the Boot ROM copies the stat register value into memory before clearing it. After booting, you can read the value of the reset status register at memory address ($r0 + 0x0038$). For more information, refer to the "Shared Memory" section of the *Booting and Configuration* appendix.

Related Information

[Shared Memory](#) on page 29-31

Functional Description of the Reset Manager

The reset manager generates reset signals to modules in the HPS and to the FPGA fabric. The following actions generate reset signals:

- Software writing a 1 to the swcoldrstreq or swwarmrstreq bits in the ctrl register. Writing either bit causes the reset controller to perform a reset sequence.
- Software writing to the mpumodrst, permodrst, per2modrst, brgmodrst, or miscmodrst module reset control registers.
- Asserting reset request signals triggers the reset controller. All external reset requests cause the reset controller to perform a reset sequence.

Multiple reset requests can be driven to the reset manager at the same time. Cold reset requests take priority over warm and debug reset requests. Higher priority reset requests preempt lower priority reset requests. There is no priority difference among reset requests within the same domain.

If a cold reset request is issued while another cold reset is already underway, the reset manager extends the reset period for all the module reset outputs until all cold reset requests are removed. If a cold reset request is issued while the reset manager is removing other modules out of the reset state, the reset manager returns those modules back to the reset state.

If a warm reset request is issued while another warm reset is already underway, the first warm reset completes before the second warm reset begins. If the second warm reset request is removed before the first warm reset completes, the warm first reset is extended to meet the timing requirements of the second warm reset request.

The nPOR pin can be used to extend the cold reset beyond what the POR voltage monitor automatically provides. The use of the nPOR pin is optional and can be tied high when it is not required.

Related Information

[Arria V GX, GT, SX, and ST Device Datasheet](#)

For information about the required duration of reset request signal assertion, refer to the [Arria V Device Datasheet](#).

Reset Sequencing

The reset controller sequences resets without software assistance. Module reset signals are asserted asynchronously and synchronously. The reset manager deasserts the module reset signals synchronous to the `osc1_clk` clock. Module reset signals are deasserted in groups in a fixed sequence. All module reset signals in a group are deasserted at the same time.

The reset manager sends a safe mode request to the clock manager to put the clock manager in safe mode, which creates a fixed and known relationship between the `osc1_clk` clock and all other clocks generated by the clock manager.

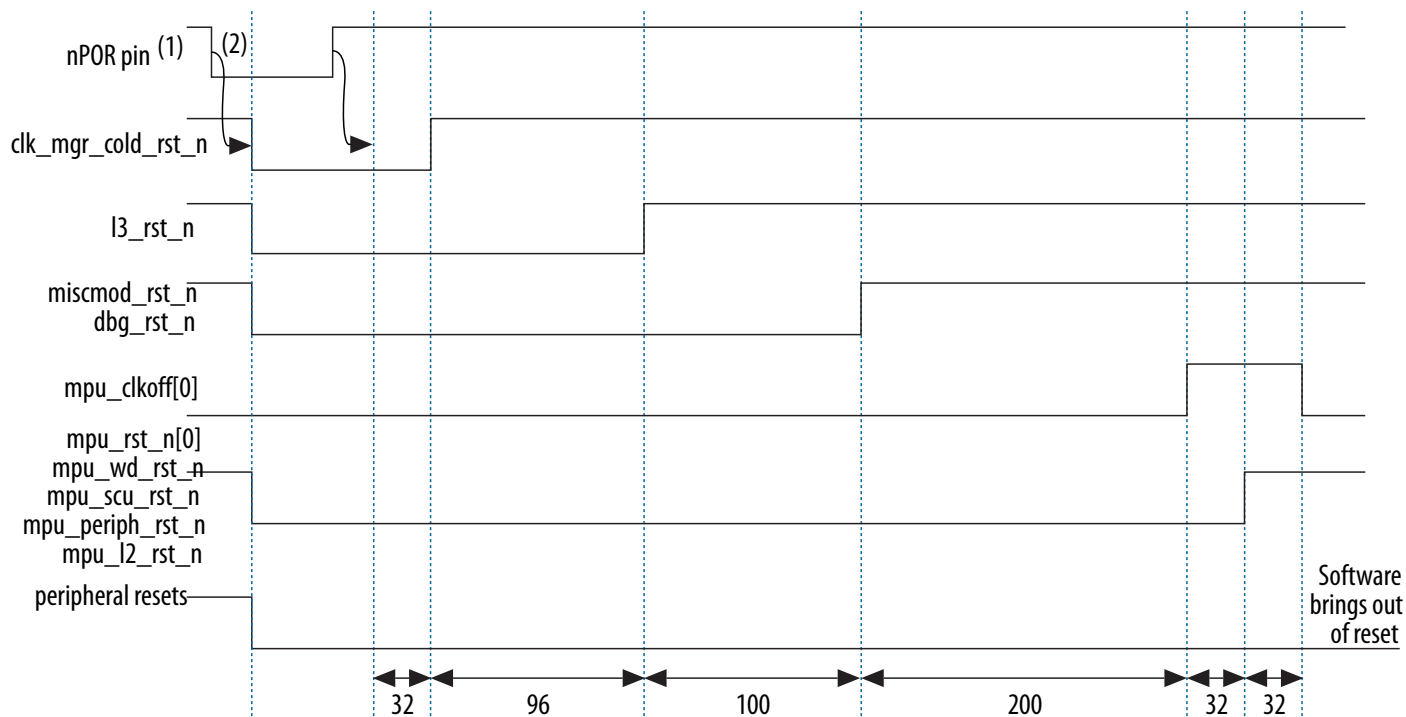
After the reset manager releases the MPU subsystem from reset, CPU1 is left in reset and CPU0 begins executing code from the reset vector address. Software is responsible for deasserting CPU1 and other resets, as shown in the MPU Group and Generated Module Resets table. Software deasserts resets by writing the `mpumodrst`, `permodrst`, `per2modrst`, `brgmodrst`, and `miscmodrst` module-reset control registers.

Software can also bypass the reset controller and generate reset signals directly through the module-reset control registers. In this case, software is responsible for asserting module reset signals, driving them for the appropriate duration, and deasserting them in the correct order. The clock manager is not typically in safe mode during this time, so software is responsible for knowing the relationship between the clocks generated by the clock manager. Software must not assert a module reset signal that would prevent software from deasserting the module reset signal. For example, software should not assert the module reset to the processor executing the software.

Table 4-10: Minimum Pulse Width

Reset Type	Value
Warm Reset	6 <code>osc1_clk</code> cycles
Cold Reset	6 <code>osc1_clk</code> cycles

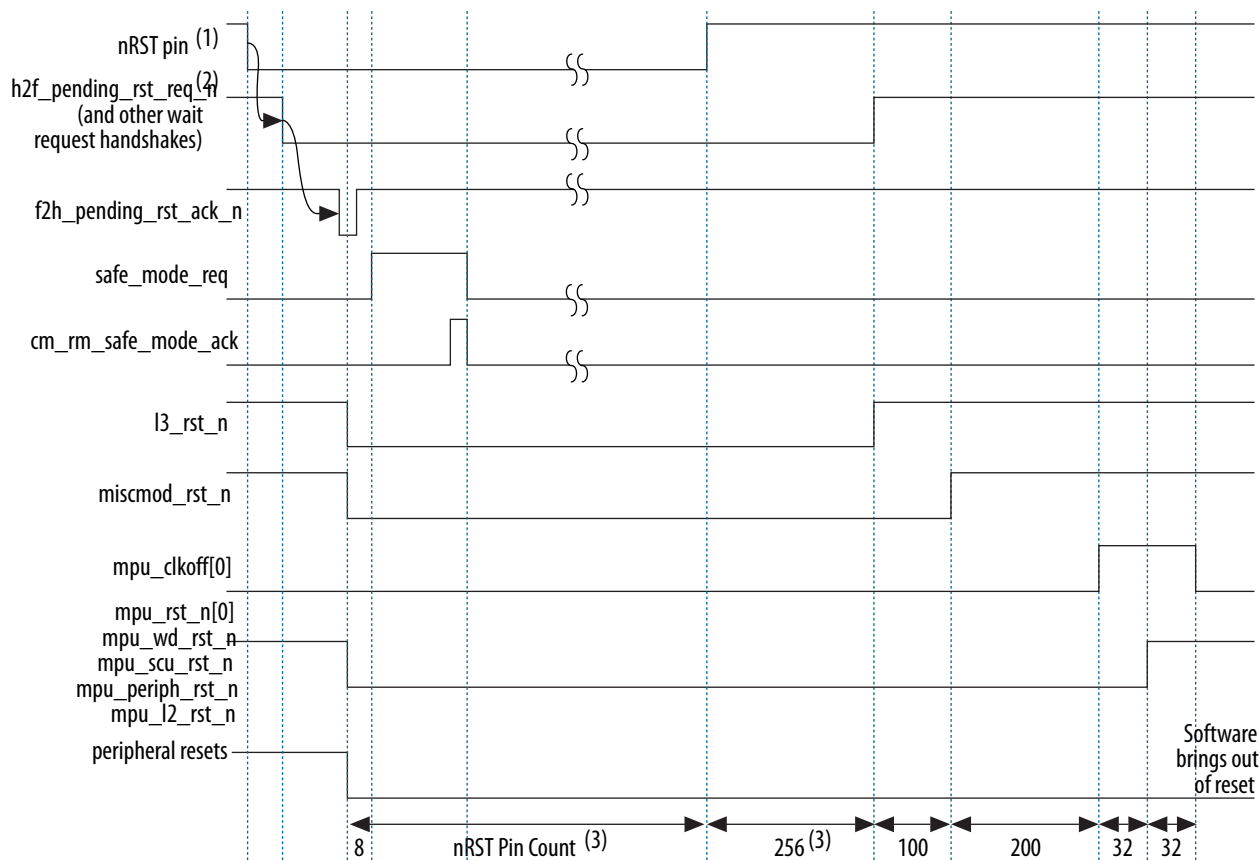
Figure 4-3: Cold Reset Timing Diagram



(1) Cold reset can be initiated from several other sources: FPGA CB, FPGA fabric, modules in the HPS, and reset pins.

(2) This dependency applies to all the reset signals.

Figure 4-4: Warm Reset Timing Diagram



(1) Cold reset can be initiated from several other sources: FPGA CB, FPGA fabric, modules in the HPS, and reset pins.

(2) When the nRSTpin count is zero, the 256 cycle stretch count is skipped and the start of the deassertion sequence is determined by the safe mode acknowledge signal or the user releasing the warm reset button, whichever occurs later.

The cold and warm reset sequences consist of different reset assertion sequences and the same deassertion sequence. The following sections describe the sequences.

Note: Cold and warm reset affect only the cpu0, and by default cpu1 is held in reset until the software running in the cpu0 releases it.

Related Information

- [Module Reset Signals](#) on page 4-5
 - [Clock Manager](#) on page 3-1
- For more information about safe mode, refer to the *Clock Manager* chapter.

Cold Reset Assertion Sequence

The following list describes the assertion steps for cold reset shown in the Cold Reset timing diagram:

1. Assert module resets
2. Wait for 32 cycles. Deassert clock manager cold reset.
3. Wait for 96 cycles (so clocks can stabilize).
4. Proceed to the “Cold and Warm Reset Deassertion Sequence” section using the following link.

Related Information

[Cold and Warm Reset Deassertion Sequence](#) on page 4-14

Warm Reset Assertion Sequence

The following list describes the assertion steps for warm reset shown in the Warm Reset Timing Diagram:

1. Optionally, handshake with the embedded trace router (ETR) and wait for acknowledge.
2. Optionally, handshake with the FPGA fabric and wait for acknowledge.
3. Optionally, handshake with the SDRAM controller, scan manager, and FPGA manager, and wait for acknowledges.
4. Assert module resets (except the MPU watchdog timer resets when the MPU watchdog timers are the only request sources).
5. Wait for 8 cycles and send a safe mode request to the clock manager.
6. Wait for the greater of the nRST pin count + 256 stretch count, or the warm reset counter, or the clock manager safe mode acknowledge, then deassert all handshakes except warm reset ETR handshake (which is deasserted by software).
7. Proceed to the “Cold and Warm Reset Deassertion Sequence” section using the following link.

Note: The nRST is a bidirectional signal that is driven out when a warm reset is generated in the chip.

Related Information

[Cold and Warm Reset Deassertion Sequence](#) on page 4-14

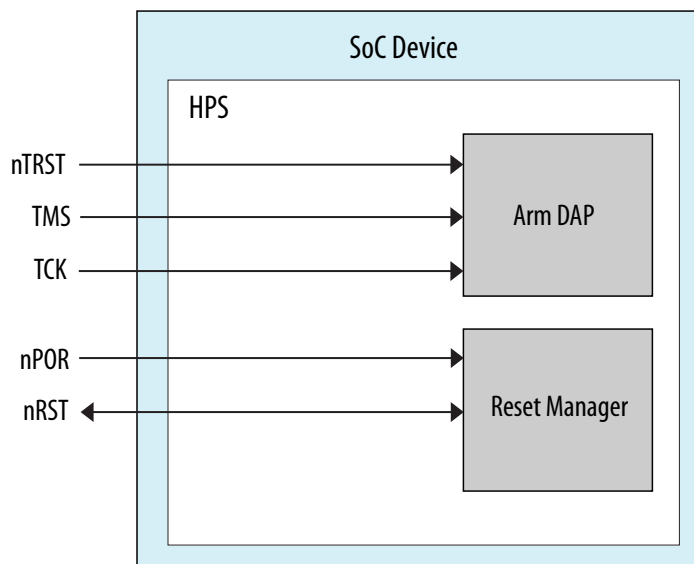
Cold and Warm Reset Deassertion Sequence

The following list describes the deassertion steps for both cold and warm reset shown in the Cold Reset Timing Diagram and Warm Reset Timing Diagram:

1. Deassert L3 reset.
2. Wait for 100 cycles. Deassert resets for miscellaneous-type and debug (cold only) modules.
3. Wait for 200 cycles. Assert mpu_clkoff for CPU0 and CPU1.
4. Wait for 32 cycles. Deassert resets for MPU modules.
5. Wait for 32 cycles. Deassert mpu_clkoff for CPU0 and CPU1.
6. Peripherals remain held in reset until software brings them out of reset.

Reset Pins

Figure 4-5: Reset Pins



The test reset (nTRST), test mode select (TMS), and test clock (TCK) pins are associated with the TAP reset domain and are used to reset the TAP controller in the DAP. These pins are not connected to the reset manager.

The nPOR and nRST pins are used to request cold and warm resets respectively. The nRST pin is an open drain output as well. A warm reset drives the nRST pin low. The amount of time the reset manager pulls nRST low is controlled by the nRST pin count field (nrstcnt) of the reset cycles count register (counts). This technique can be used to reset external devices (such as external memories) connected to the HPS.

Reset Effects

The following list describes how reset affects HPS logic:

- The TAP reset domain ignores warm reset.
- The debug reset domain ignores warm reset.
- System reset domain cold resets ignore warm reset.
- Each module defines reset behavior individually.

Altering Warm Reset System Response

Registers in the clock manager, system manager, and reset manager control how warm reset affects the HPS. You can control the impact of a warm reset on the clocks and I/O elements.

Intel strongly recommends using provided libraries to configure and control this functionality.

The default warm reset behavior takes all clocks and I/O elements through a cold reset response. As your software becomes more stable or for debug purposes, you can alter the system response to a warm reset. The following suggestions provide ways to alter the system response to a warm reset. None of the register bits that control these items are affected by warm reset.

- Boot from on-chip RAM—enables warm boot from on-chip RAM instead of the boot ROM. When enabled, the boot ROM code validates the RAM code and jumps to it, making no changes to clocks or any other system settings prior to executing user code from on-chip RAM.
- Disable safe mode on warm reset—allows software to transition through a warm reset without affecting the clocks. Because the boot ROM code indirectly configures the clock settings after warm reset, Intel recommends that safe mode should only be disabled when the HPS is not booting from a flash device.
- Disable safe mode on warm reset for the debug clocks—keeps the debug clocks from being affected by the assertion of safe mode request on a warm reset. This technique allows fast debug clocks, such as trace, to continue running through a warm reset. When enabled, the clock manager configures the debug clocks to their safe frequencies to respond to a safe mode request from the reset manager on a warm reset. Disable safe mode on warm reset for the debug clocks only when you are running the debug clocks off the main PLL VCO and you are certain the main PLL cannot be impacted by the event which caused the warm reset.
- Use the `osc1_clk` clock for debug control—keeps the debug base clock (main PLL C2 output) always bypassed to the `osc1_clk` external clock, independent of other clock manager settings. When implemented, disabling safe mode on warm reset for the debug clocks has no effect.

Related Information

[Clock Manager](#) on page 3-1

For more information about safe mode, refer to the *Clock Manager* chapter.

Reset Handshaking

The reset manager participates in several reset handshaking protocols to ensure other modules are safely reset.

Before issuing a warm reset, the reset manager performs a handshake with several modules to allow them to prepare for a warm reset. The handshake logic ensures the following conditions:

- Optionally the ETR master has no pending master transactions to the L3 interconnect
- Optionally preserve SDRAM contents during warm reset by issuing self-refresh mode request
- FPGA manager stops generating configuration clock
- Scan manager stops generating JTAG and I/O configuration clocks
- Warns the FPGA fabric of the forthcoming warm reset

Similarly, the handshake logic associated with ETR also occurs during the debug reset to ensure that the ETR master has no pending master transactions to the L3 interconnect before the debug reset is issued. This action ensures that when ETR undergoes a debug reset, the reset has no adverse effects on the system domain portion of the ETR.

Reset Manager Address Map and Register Definitions

The address map and register definitions for the HPS-FPGA bridge consist of the following regions:

- Reset Manager Module

2021.05.07

av_5v4



Subscribe



Send Feedback

The FPGA manager in the hard processor system (HPS) manages and monitors the FPGA portion of the system on a chip (SoC) device. The FPGA manager can configure the FPGA fabric from the HPS, monitor the state of the FPGA, and drive or sample signals to or from the FPGA fabric.

Features of the FPGA Manager

The FPGA manager provides the following functionality and features:

- Full configuration and partial reconfiguration of the FPGA portion of the SoC device
- Drives 32 general-purpose output signals to the FPGA fabric
- Receives 32 general-purpose input signals from the FPGA fabric
- Receives two boot handshaking input signals from the FPGA fabric (used when the HPS boots from the FPGA)
- Monitors the FPGA configuration and power status
- Generates interrupts based on the FPGA status changes
- Can reset the FPGA

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

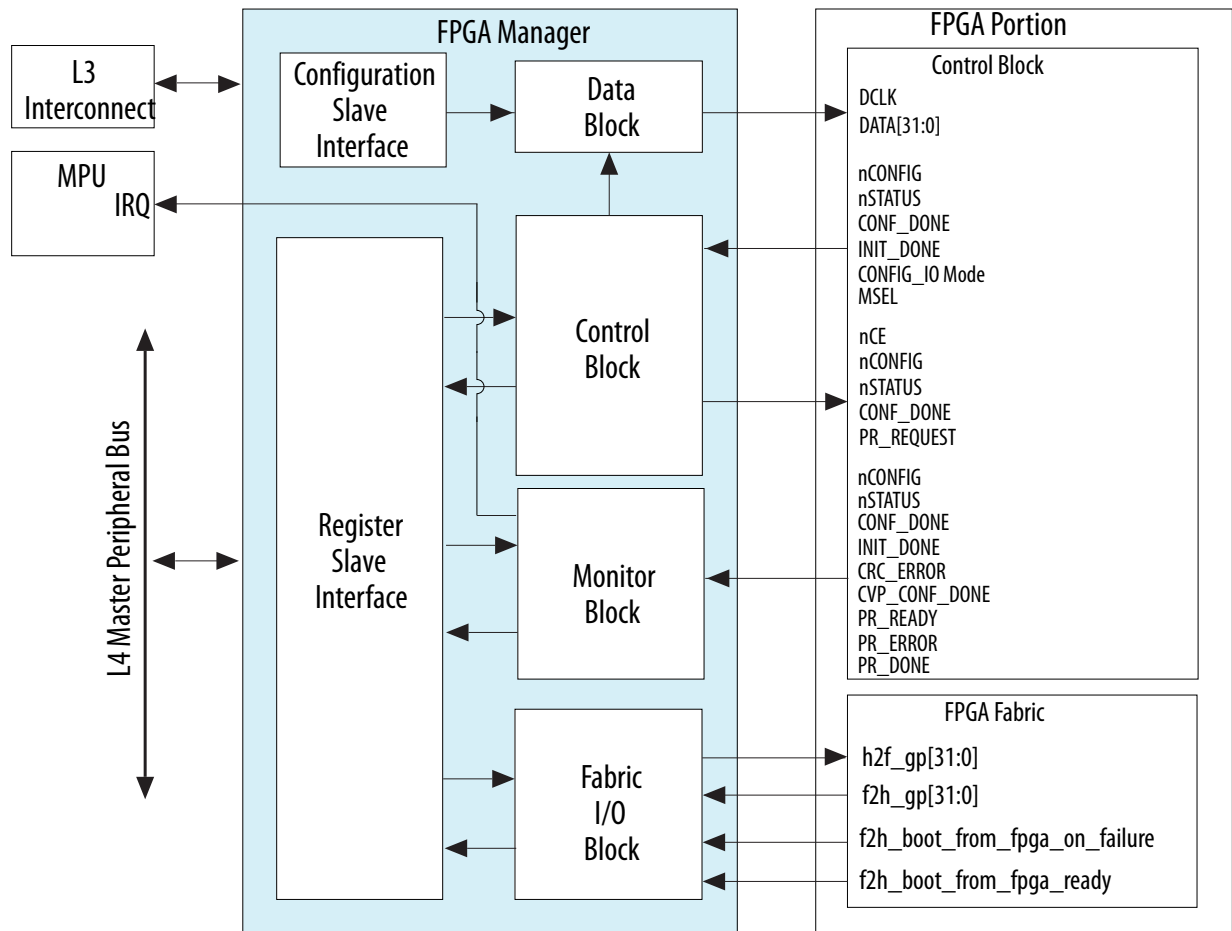
*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

FPGA Manager Block Diagram and System Integration

Figure 5-1: FPGA Manager Block Diagram



The register slave interface connects to the level 4 (L4) master peripheral bus for control and status register (CSR) access. The configuration slave interface connects to the level 3 (L3) interconnect for the microprocessor unit (MPU) subsystem or other masters to write the FPGA configuration image to the FPGA control block (CB) when configuring the FPGA portion of the SoC device.

The general-purpose I/O and boot handshake input interfaces connect to the FPGA fabric. The FPGA manager also connects to the FPGA CB signals to monitor and control the FPGA portion of the device.

The FPGA manager consists of the following blocks:

- Configuration slave interface—accepts and transfers the configuration image to the data interface.
- Register slave interface—accesses the CSRs in the FPGA manager.
- Data—accepts the FPGA configuration image from the configuration slave interface and sends it to the FPGA CB.
- Control—controls the FPGA CB.
- Monitor—monitors the configuration signals in the FPGA CB and sends interrupts to the MPU subsystem.
- Fabric I/O—reads and writes signals from or to the FPGA fabric.

Functional Description of the FPGA Manager

FPGA Manager Building Blocks

The FPGA manager has the two blocks - fabric I/O and monitor.

Related Information

[FPGA Manager Address Map and Register Definitions](#) on page 5-9

Fabric I/O

The fabric I/O block contains the following registers to allow simple low-latency communication between the HPS and the FPGA fabric:

- General-purpose input register (gpi)
- General-purpose output register (gpo)
- Boot handshaking input register (misci)

These registers are only valid when the FPGA is in user mode. Reading from these registers while the FPGA is not in user mode provides undefined data.

The 32 general-purpose input signals from the FPGA fabric are read by reading the gpi register using the register slave interface. The 32 general-purpose output signals to the FPGA fabric are generated from writes to the gpo register. For more information about FPGA manager registers, refer to [FPGA Manager Address Map and Register Definitions](#) on page 5-9.

The boot handshake input signals from the FPGA fabric are read by reading the misci register. The f2h_boot_from_fpga_ready signal indicates that the FPGA fabric is ready to send preloader information to the boot ROM. The f2h_boot_from_fpga_on_failure signal serves as a fallback in the event that the boot ROM code fails to boot from the primary boot flash device. In this case, the boot ROM code checks these two handshaking signals to determine if it should use the boot code hosted in the FPGA memory as the next stage in the boot process.

There is no interrupt support for this block.

Related Information

[FPGA Manager Address Map and Register Definitions](#) on page 5-9



Monitor

The monitor block is an instance of the SynopsysGPIO IP (DW_apb_gpio), which is a separate instance of the IP that comprises the three HPS GPIO interfaces. The monitor block connects to the configuration signals in the FPGA. This block monitors key signals related to FPGA configuration such as INIT_DONE, CRC_ERROR, and PR_DONE. Software configures the monitor block through the register slave interface, and can either poll FPGA signals or be interrupted. The **mon** address map within the FPGA manager register address map contains the monitor registers. For more information about FPGA manager registers, refer to [FPGA Manager Address Map and Register Definitions](#) on page 5-9

You can program the FPGA manager to treat any of the monitor signals as interrupt sources. Independent of the interrupt source type, the monitor block always drives an active-high level interrupt to the MPU. Each interrupt source can be of the following types:

- Active-high level
- Active-low level
- Rising edge
- Falling edge

FPGA Configuration

You can configure the FPGA using an external device or through the HPS. This section highlights configuring the FPGA through the HPS.

The FPGA CB uses the FPGA mode select (MSEL) pins to determine which configuration scheme to use. The MSEL pins must be tied to the appropriate values for the configuration scheme. The table below lists supported MSEL values when the FPGA is configured by the HPS.

Table 5-1: Configuration Schemes for FPGA Configuration by the HPS

Configuration Scheme	Compression Feature	Design Security Feature	POR Delay ⁽¹⁰⁾	MSEL[4..0] ⁽¹¹⁾	cfgwidth	cdratio	Supports Partial Reconfiguration
FPP ×16	Disabled	AES Disabled	Fast	00000	0	1	Yes
			Standard	00100	0	1	Yes
	Disabled	AES Enabled	Fast	00001	0	2	Yes
			Standard	00101	0	2	Yes
	Enabled	Optional	Fast	00010	0	4	Yes
			Standard	00110	0	4	Yes
FPP ×32 ⁽¹³⁾	Disabled	AES Disabled	Fast	01000	1	1	No
			Standard	01100	1	1	No
	Disabled	AES Enabled	Fast	01001	1	4	No
			Standard	01101	1	4	No
	Enabled	Optional ⁽¹²⁾	Fast	01010	1	8	No
			Standard	01110	1	8	No

HPS software sets the clock-to-data ratio field (*cdratio*) and configuration data width bit (*cfgwidth*) in the control register (*ctrl*) to match the MSEL pins. The *cdratio* field and *cfgwidth* bit must be set before the start of configuration.

The FPGA manager connects to the configuration logic in the FPGA portion of the device using a mode similar to how external logic (for example, MAX II or an intelligent host) configures the FPGA in fast

⁽¹⁰⁾ For information about POR delay, refer to the Configuration, Design Security, and Remote System Upgrades in Arria V Devices.

⁽¹¹⁾ Other MSEL values are allowed when the FPGA is configured from a non-HPS source. For information, refer to the Configuration, Design Security, and Remote System Upgrades in Arria V Devices.

⁽¹²⁾ You can select to enable or disable this feature.

⁽¹³⁾ When the FPGA is configured through the HPS, then FPPx32 is supported. Otherwise, if the FPGA is configured from a non-HPS (external) source, then FPPx32 is not supported. For more information refer to the Configuration, Design Security, and Remote System Upgrades in Arria V Devices.

passive parallel (FPP) mode. FPGA configuration through the HPS supports all the capabilities of FPP mode, including the following items:

- FPGA configuration
- Partial FPGA reconfiguration
- FPGA I/O configuration, followed by PCI Express (PCIe®) configuration of the remainder of FPGA
- External single event upset (SEU) scrubbing
- Decompression
- Advanced Encryption Standard (AES) encryption
- FPGA DCLK clock used for initialization phase clock

Configuring the FPGA portion of the SoC device comprises the following phases:

1. Power up phase
2. Reset phase
3. Configuration phase
4. Initialization phase
5. User mode

The FPGA Manager can be configured to accept configuration data directly from the MPU or the DMA engine. Either the processor or the DMA engine moves data from memory to the FPGA Manager data image register space `img_data_w`. The L4 interconnect allocates a 4 KB region for image data. It is not necessary to increment the address when writing the image data because all accesses within the 4 KB image data region is transferred to the configuration logic.

Related Information

- [Configuration, Design Security, and Remote System Upgrades in Arria V Devices](#)
For more information about configuring the FPGA through the HPS, refer to the "Configuration, Design Security, and Remote System Upgrade in Arria V Devices" chapter in the *Arria V Device Handbook Volume 1: Device Interfaces and Integration*.
- [Bootting and Configuration](#) on page 29-1

Power Up Phase

In this phase, the VCC is ramping up and has yet to reach normal levels. This phase completes when the on-chip voltage detector determines that the VCC has reached normal levels.

Reset Phase

The FPGA manager resets the FPGA portion of the SoC device when the FPGA configuration signal (`nCONFIG`) is driven low. The HPS configures the FPGA by writing a 1 to the `nconfigpull` bit of the `ctrl` register. This action causes the FPGA portion of the device to reset and perform the following actions:

1. Clear the FPGA configuration RAM bits
2. Tri-state all FPGA user I/O pins
3. Pull the `nSTATUS` and `CONF_DONE` pins low
4. Use the FPGA CB to read the values of the `MSEL` pins to determine the configuration scheme

The `nconfigpull` bit of the `ctrl` register needs to be set to 0 when the FPGA has successfully entered the reset phase. Setting the bit releases the FPGA from the reset phase and transitions to the configuration phase.

Note: You must set the `cdratio` and `cfgwidth` bits of the `ctrl` register appropriately before the FPGA enters the reset phase.

Configuration Phase

To configure the FPGA using the HPS, software sets the `axicfgen` bit of the `ctrl` register to 1. Software then sends configuration data to the FPGA by writing data to the write data register (`data`) in the FPGA manager module configuration data address map. Software polls the `CONF_DONE` pin by reading the `gpio_instatus` register to determine if the FPGA configuration is successful. When configuration is successful, software sets the `axicfgen` bit of the `ctrl` register to 0. The FPGA user I/O pins are still tri-stated in this phase.

After successfully completing the configuration phase, the FPGA transitions to the initialization phase. To delay configuring the FPGA, set the `confdonepull` bit of the `ctrl` register to 1.

Related Information

[Booting and Configuration](#) on page 29-1

Initialization Phase

In this phase, the FPGA prepares to enter user mode. The internal oscillator in the FPGA portion of the device is the default clock source for the initialization phase. Alternatively, the configuration image can specify the `CLKUSR` or the `DCLK` pins as the clock source. The alternate clock source controls when the FPGA enters user mode.

If `DCLK` is selected as the clock source, software uses the `DCLK` count (`dcclkcnt`) register to drive `DCLK` pulses to the FPGA. Writing to the `cnt` field of the `dcclkcnt` register triggers the FPGA manager to generate the specified number of `DCLK` pulses. When all of the `DCLK` pulses have been sent, the `dcntdone` bit of the `DCLK` status (`dcclkstat`) register is set to 1. Software polls the `dcntdone` bit to know when all of the `DCLK` pulses have been sent.

Note: Before another write to the `dcclkcnt` register, software needs to write a value of 1 to the `dcntdone` bit to clear the done state.

The FPGA user I/O pins are still tri-stated in this phase. When the initialization phase completes, the FPGA releases the optional `INIT_DONE` pin and an external resistor pulls the pin high.

User Mode

The FPGA enters the user mode after exiting the initialization phase. The FPGA user I/O pins are no longer tri-stated in this phase and the configured soft logic in the FPGA becomes active.

The FPGA remains in user mode until the `nCONFIG` pin is driven low. If the `nCONFIG` pin is driven low, the FPGA reenters the reset phase. The internal oscillator is disabled in user mode, but is enabled as soon as the `nCONFIG` pin is driven low.

Related Information

- [Configuration, Design Security, and Remote System Upgrades in Arria V Devices](#)
For more information about configuring the FPGA through the HPS, refer to the "Configuration, Design Security, and Remote System Upgrade in Arria V Devices" chapter in the *Arria V Device Handbook Volume 1: Device Interfaces and Integration*.
- [Booting and Configuration](#) on page 29-1

FPGA Status

Configuration signals from the FPGA CB such as INIT_DONE, CRC_ERROR and PR_DONE are monitored by the FPGA Manager. Software configures the monitor block through the register slave interface, and can either poll FPGA signals or be interrupted. Monitored signals can be read through the `imgcfg_stat` register as well as the `intr_masked_status` register. Each of the monitored signals can generate an interrupt to the MPU global interrupt controller. Each interrupt source can be enabled and the polarity of the signals generating the interrupt can also be selected through the `intr_mask` and `intr_polarity` registers in the FPGA Manager.

Error Message Extraction

Cyclic redundancy check (CRC) errors from the FPGA fabric are monitored by the FPGA Manager. Upon assertion of a CRC error signal from the FPGA, the FPGA Manager extracts information about the error including:

- Error syndrome
- Error location
- Error type

A CRC error interrupt from the FPGA manager can be enabled through software. Software can then extract the CRC error information from the error message register (EMR) data interface. The number of valid error information bits in the EMR data registers depends on the specific FPGA device.

Boot Handshake

There are two input signals from the FPGA to control HPS boot from the FPGA. Both are synchronized within the FPGA Manager. Boot software reads these signals before accessing a boot image in the FPGA. The following table describes the functionality of these signals.

Signal	Description
f2h_boot_from_fpga_on_failure	Indicates whether a fallback preloader image is available in the FPGA on-chip RAM at memory location 0x0. The fallback preloader image is used only if the HPS boot ROM does not find a valid preloader image in the selected flash memory device. It is an active high signal which the bootrom polls when all the preloaders fail to load. This signal is driven low when the FPGA is not configured and it is up to you to drive it high in your user design.
f2h_boot_from_fpga_ready	Indicates a preloader image is available in an FPGA on-chip RAM at memory location 0x0 and it is ready to be accessed. It is an active high signal which the bootrom polls to determine when the FPGA is configured and the memory located at offset 0x0 from the F2H bridge is ready to be written to. When the FPGA is not configured the hardware drives this signal low and it is up to you to drive it high when the memory in the FPGA is ready to accept memory mapped transactions.

General Purpose I/O

Thirty-two general purpose inputs and thirty-two general purpose outputs are provided to the FPGA and are controlled through registers in the FPGA Manager.

No interrupts are generated through the input pins. All inputs are synchronized within the FPGA Manager. Output signals should be synchronized in the FPGA.

Clock

The FPGA manager has two clock input signals which are asynchronous to each other. The clock manager generates these two clocks:

- `cfg_clk`—the configuration slave interface clock input and also the DCLK output reference for FPGA configuration. Enable this clock in the clock manager only when configuration is active or when the configuration slave interface needs to respond to master requests.
- `l4_mp_clk`—the register slave interface clock.

Related Information

[Clock Manager](#) on page 3-1

Reset

The FPGA manager has the `fpga_manager_rst_n` reset signal. The reset manager drives this signal to the FPGA manager on a cold or warm reset. All distributed reset signals in the FPGA manager are asserted asynchronously at the same time and deasserted synchronously to their associated clocks.

Related Information

[Reset Manager](#) on page 4-1

FPGA Manager Address Map and Register Definitions

The address map and register definitions for the FPGA Manager consist of the following regions:

- FPGA Manager Module
- FPGA Manager Module Configuration Data

Related Information

[Introduction to the Hard Processor System](#) on page 2-1

2021.05.07

av_5v4



Subscribe



Send Feedback

The system manager in the hard processor system (HPS) contains memory-mapped control and status registers (CSRs) and logic to control system level functions as well as other modules in the HPS.

The system manager connects to the following modules in the HPS:

- Direct memory access (DMA) controller
- Ethernet media access controllers (EMAC0 and EMAC1)
- Microprocessor unit (MPU) subsystem
- NAND flash controller
- Secure Digital/MultiMediaCard (SD/MMC) controller
- Quad serial peripheral interface (SPI) flash controller
- USB 2.0 On-The-Go (OTG) controllers (USB0 and USB1)
- Watchdog timers

Features of the System Manager

Software accesses the CSRs in the system manager to control and monitor various functions in other HPS modules that require external control signals. The system manager connects to these modules to perform the following functions:

- Sends pause signals to pause the watchdog timers when the processors in the MPU subsystem are in debug mode
- Selects the EMAC level 3 (L3) master signal options.
- Freezes the I/O pins after the HPS comes out of cold reset and during serial configuration.
- Selects the SD/MMC controller clock options and L3 master signal options.
- Selects the NAND flash controller bootstrap options and L3 master signal options.
- Selects USB controller L3 master signal options.
- Provides control over the DMA security settings when the HPS exits from reset.
- Provides boot source and clock source information that can be read during the boot process.
- Provides the capability to enable or disable an interface to the FPGA.
- Routes parity failure interrupts from the L1 caches to the Global Interrupt Controller.
- Sends error correction code (ECC) enable signals to all HPS modules with ECC-protected RAM.
- Provides the capability to inject errors during testing in the MPU L2 ECC-protected RAM.

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

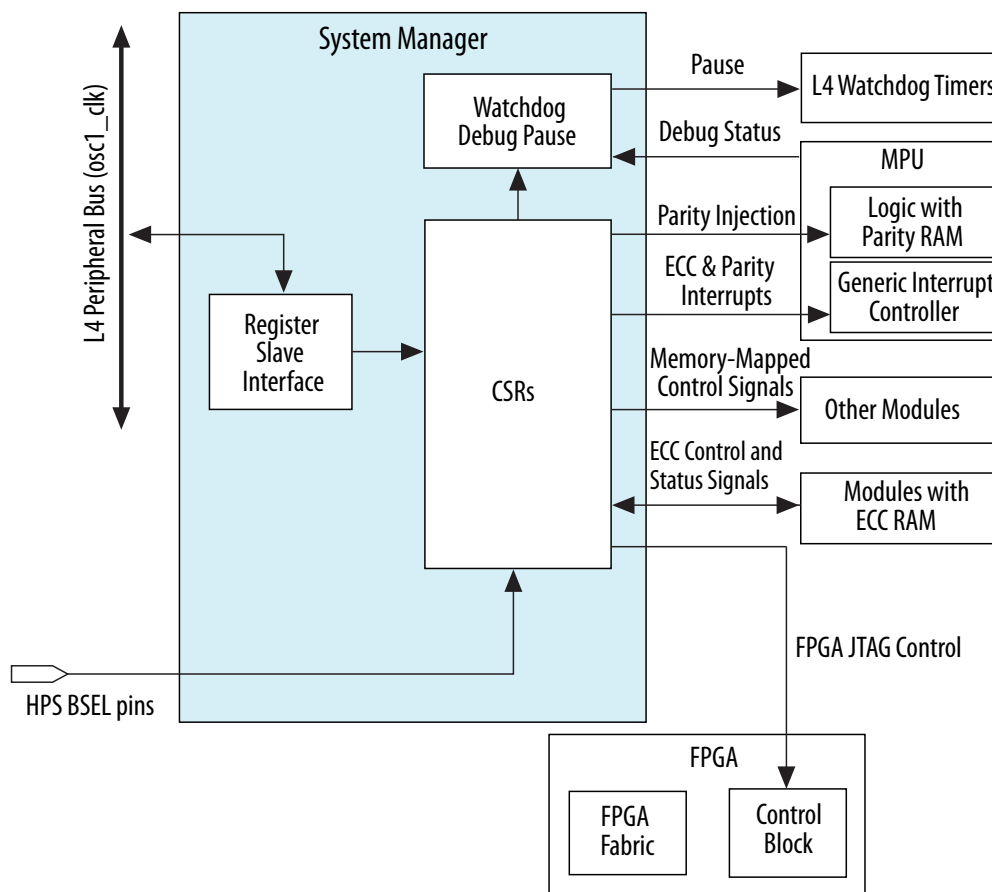
ISO
9001:2015
Registered

ALTERA
now part of Intel

System Manager Block Diagram and System Integration

The system manager connects to the level 4 (L4) bus through a slave interface. The CSRs connect to signals in the FPGA and other HPS modules.

Figure 6-1: System Manager Block Diagram



The system manager consists of the following:

- CSRs—Provide memory-mapped access to control signals and status for the following HPS modules:
 - EMACs
 - Debug core
 - SD/MMC controller
 - NAND controller
 - USB controllers
 - DMA controller
 - System interconnect
 - ECC memory interfaces for the following peripherals:
 - USB controllers
 - SD/MMC controller
 - Ethernet MACs
 - DMA controller
 - NAND flash controller
 - On-chip RAM
- Slave port interface—provides access to system manager CSRs for connected masters.
- Watchdog debug pause—accepts the debug mode status from the MPU subsystem and pauses the L4 watchdog timers.
- Reset Manager— system manager receives the reset signals from reset manager.

Functional Description of the System Manager

The system manager serves the following purposes:

- Provides software access to boot configuration and system information
- Provides software access to control and status signals in other HPS modules
- Provides combined ECC status and interrupt from other HPS modules with ECC-protected RAM
- Enables and disables HPS peripheral interfaces to the FPGA
- Provides eight registers that software can use to pass information between boot stages

Boot Configuration and System Information

The system manager provides boot configuration information through the `bootinfo` register. Sampled value of the HPS boot select (BSEL) pins are available to the Boot ROM software.

The boot source is determined by the BSEL pins.

Related Information

[Booting and Configuration](#) on page 29-1

Additional Module Control

Each module in the HPS has its own CSRs, providing access to the internal state of the module. The system manager provides registers for additional module control and monitoring. To fully control each module,

you must program both the peripheral's CSR and its corresponding CSR in the System Manager. This section describes how to configure the system manager CS for each module.

DMA Controller

The security state of the DMA controller is controlled by the manager thread security (`mgr_ns`) and interrupt security (`irq_ns`) bits of the DMA register.

The `periph_ns` register bits determine if a peripheral request interface is secure or non-secure.

Note: The `periph_ns` register bits must be configured before the DMA is released from global reset.

Related Information

- [DMA Controller](#) on page 16-1
- [System Manager Address Map and Register Definitions](#) on page 6-9

NAND Flash Controller

The bootstrap control register (`nand_bootstrap`) modifies the default behavior of the NAND flash controller after reset. The NAND flash controller samples the bootstrap control register bits when it comes out of reset.

The following `nand_bootstrap` register bits control configuration of the NAND flash controller:

- Bootstrap inhibit initialization bit (`noinit`)—inhibits the NAND flash controller from initializing when coming out of reset, and allows software to program all registers pertaining to device parameters such as page size and width.
- Bootstrap 512-byte device bit (`page512`)—informs the NAND flash controller that a NAND flash device with a 512-byte page size is connected to the system.
- Bootstrap inhibit load block 0 page 0 bit (`no_loadb0p0`)—inhibits the NAND flash controller from loading page 0 of block 0 of the NAND flash device during the initialization procedure.
- Bootstrap two row address cycles bit (`tworowaddr`)—informs the NAND flash controller that only two row address cycles are required instead of the default three row address cycles.

You can use the system manager's `nanad_l3master` register to control the following signals:

- `ARPROT`
- `AWPROT`
- `ARDOMAIN`
- `AWDOMAIN`
- `ARCACHE`
- `AWCACHE`

These bits define the cache attributes for the master transactions of the DMA engine in the NAND controller.

Note: Register bits must be accessed only when the master interface is guaranteed to be in an inactive state.

Related Information

- [NAND Flash Controller](#) on page 13-1
- [System Manager Address Map and Register Definitions](#) on page 6-9

EMAC

You can program the `emac_global` register to select either `emac_ptp_clk` from the Clock Manager or `f2s_ptp_ref_clk` from the FPGA fabric as the source of the IEEE 1588 reference clock for each EMAC.

You can use the system manager's `l3master` register to control the EMAC's ARCACHE and AWCACHE signals, by setting or clearing the (`arcache`, `awcache`) bits. These bits define the cache attributes for the master transactions of the DMA engine in the EMAC controllers.

Note: Register bits must be accessed only when the master interface is guaranteed to be in an inactive state.

The `phy_intf_sel` bit is programmed to select between a GMII (MII), RGMII or RMII PHY interface when the peripheral is released from reset. The `ptp_ref_sel` bit selects if the timestamp reference is internally or externally generated. The `ptp_ref_sel` bit must be set to the correct value before the EMAC core is pulled out of reset.

Related Information

- [Clock Manager](#) on page 3-1
- [Ethernet Media Access Controller](#) on page 17-1

USB 2.0 OTG Controller

The `usb*_l3master` registers in the system manager control the `HPROT` and `HAUSER` fields of the USB master port of the USB 2.0 OTG Controller.

Note: Register bits should be accessed only when the master interface is guaranteed to be in an inactive state.

Related Information

[USB 2.0 OTG Controller](#) on page 18-1

SD/MMC Controller

The `sdmmc_l3master` register in the system manager controls the `HPROT` and `HAUSER` fields of the SD/MMC master port.

Note: Register bits should be accessed only when the master interface is guaranteed to be in an inactive state.

You can program software to select the clock's phase shift for `cclk_in_drv` and `cclk_in_sample` by setting the drive clock phase shift select (`drvsel`) and sample clock phase shift select (`smp1sel`) bits of the `sdmmc` register in the system manager.

Related Information

[SD/MMC Controller](#) on page 14-1

Watchdog Timer

The system manager controls the watchdog timer behavior when the CPUs are in debug mode. The system manager sends a pause signal to the watchdog timers depending on the setting of the debug mode bits of the L4 watchdog debug register (`wddbg`). Each watchdog timer built into the MPU subsystem is paused when its associated CPU enters debug mode.

Related Information[Watchdog Timer](#) on page 24-1

Boot ROM Code

Registers in the system manager control whether the boot ROM code configures the pin multiplexing for boot pins after a warm reset. Set the warm-reset-configure-pin-multiplex for boot pins bit (`warmrstcfg-pinmux`) of the boot ROM code register to enable or disable this control.

Note: The boot ROM code always configures the pin multiplexing for boot pins after a cold reset.

Registers in the system manager also control whether the boot ROM code configures the I/O pins used during the boot process after a warm reset. Set the warm reset configure I/Os for boot pins bit (`warmrstcfgio`) of the `ctrl` register to enable or disable this control.

Note: By default, the boot ROM code always configures the I/O pins used by boot after a cold reset.

When CPU1 is released from reset and the boot ROM code is located at the CPU1 reset exception address (for a typical case), the boot ROM reset handler code reads the address stored in the CPU1 start address register (`cpu1startaddr`) and passes control to software at that address.

There can be up to four preloader images stored in flash memory. The (`initswlastld`) register contains the index of the preloader's last image that is loaded in the on-chip RAM.

The boot ROM software state register (`bootromswstate`) is a 32-bit general-purpose register reserved for the boot ROM.

The following warmram related registers are used to configure the warm reset from on-chip RAM feature and must be written by software prior to the warm reset occurring.

Table 6-1: The warmram Registers

Register	Name	Purpose
<code>enable</code>	Enable	Controls whether the boot ROM attempts to boot from the contents of the on-chip RAM on a warm reset.
<code>datastart</code>	Data start	Contains the byte offset of the warm boot CRC validation region in the on-chip RAM. The offset must be word-aligned to an integer multiple of four.
<code>length</code>	Length	Contains the length in bytes of the region in the on-chip RAM available for warm boot CRC validation.
<code>execution</code>	Execution offset	Contains a 16-bit offset into the on-chip RAM that the boot code jumps to if the CRC validation succeeds. The boot ROM appends 0xFFFF to the upper 16-bits of this 32-bit register value when it is read.
<code>crc</code>	Expected CRC	Contains the expected CRC of the region in the on-chip RAM.



The number of wait states applied to the boot ROM's read operation is determined by the wait state bit (`waitstate`) of the `ctrl` register. After the boot process, software might require reading the code in the boot ROM. If software has changed the clock frequency of the `l3_main_clk` after reset, an additional wait state is necessary to access the boot ROM. Set the `waitstate` bit to add an additional wait state to the read access of the boot ROM. The enable safe mode warm reset update bit controls whether the wait state bit is updated during a warm reset.

L3 Interconnect

The System Manager provides remap bits to the L3 interconnect. These bits can remap the Boot ROM and the On-chip RAM.

FPGA Interface Enables

The system manager can enable or disable interfaces between the FPGA and HPS.

The global interface bit (`intf`) of the global disable register (`gbl`) disables all interfaces between the FPGA and HPS.

Note: Ensure that the FPGA is configured before enabling the interfaces and that all interfaces between the FPGA and HPS are inactive before disabling them.

You can program the individual disable register (`indiv`) to disable the following interfaces between the FPGA and HPS:

You can program the FPGA interface enable registers (`fpga_intf_en_*`) to disable the following interfaces between the FPGA and HPS:

- Reset request interface
- JTAG enable interface
- I/O configuration interface
- Boundary scan interface
- Debug interface
- Trace interface
- System Trace Macrocell (STM) interface
- Cross-trigger interface (CTI)
- QSPI interface
- SD/MMC interface
- SPI Master interface
- SPI Slave interface
- EMAC interface
- UART interface

ECC and Parity Control

The system manager can enable or disable ECC for each of the following HPS modules with ECC-protected RAM:

- MPU L2 cache data RAM
- On-chip RAM
- USB 2.0 OTG controller (USB0 and USB1) RAM
- EMAC (EMAC0 and EMAC1) RAM
- DMA controller RAM
- NAND flash controller RAM
- Quad SPI flash controller RAM
- SD/MMC controller RAM
- DDR interfaces

The system manager can inject single-bit or double-bit errors into the MPU L2 ECC memories for testing purposes. Set the bits in the appropriate memory enable register to inject errors. For example, to inject a single bit ECC error, set the `injs` bit of the `mpu_ctl_l2_eccregister`.

Note: The injection request is edge-sensitive, meaning that the request is latched on 0 to 1 transitions on the injection bit. The next time a write operation occurs, the data is corrupted, containing either a single or double bit error as selected. When the data is read back, the ECC logic detects the single or double bit error appropriately. The injection request cannot be cancelled, and the number of injections is limited to once every five MPU cycles.

The system manager can also inject parity failures into the parity-protected RAM in the MPU L1 to test the parity failure interrupt handler. Set the bits of the parity fail injection register (`parityinj`) to inject parity failures.

Note: Injecting parity failures into the parity-protected RAM in the MPU L1 causes the interrupt to be raised immediately. There is no actual error injected and the data is not corrupted. Furthermore, there is no need for a memory operation to actually be performed for the interrupt to be raised.

Preloader Handoff Information

The system manager provides eight 32-bit registers to store handoff information between the preloader and the operating system. The preloader can store any information in these registers. These register contents have no impact on the state of the HPS hardware. When the operating system kernel boots, it retrieves the information by reading the preloader to OS handoff information register array. These registers are reset only by a cold reset.

Clocks

The system manager is driven by a clock generated by the clock manager.

Related Information

[Clock Manager](#) on page 3-1

Resets

The system manager receives two reset signals from the reset manager. The `sys_manager_rst_n` signal is driven on a cold or warm reset and the `sys_manager_cold_rst_n` signal is driven only on a cold reset.

This function allows the system manager to reset some CSR fields on either a cold or warm reset and others only on a cold reset.

Related Information

[Reset Manager](#) on page 4-1

System Manager Address Map and Register Definitions

The address map and register definitions for the HPS-FPGA bridges consist of the following regions:

- System Manager Module

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in the *Arria V Device Handbook, Volume 3*.
- <https://www.intel.com/content/www/us/en/programmable/hps/arria-v/hps.html>

2021.05.07

av_5v4



Subscribe



Send Feedback

The scan manager is used to configure and manage the HPS I/O pins, and communicate with the FPGA JTAG test access port (TAP) controller. The scan manager drives the HPS I/O scan chains to configure the I/O bank properties before the pins are used by the peripherals in HPS. The scan manager can also optionally communicate with the FPGA JTAG TAP controller to send commands for purposes such as managing cyclic redundancy check (CRC) errors detected by the FPGA control block. When the scan manager communicates with the FPGA JTAG TAP controller, input on the FPGA JTAG pins is ignored.

The scan manager contains an Arm JTAG Access Port (JTAG-AP). The JTAG-AP implements a multiple scan-chain JTAG master interface. One scan chain connects to the FPGA JTAG and uses the standard JTAG signals. Four other scan chains connect to the HPS I/O banks, using the JTAG clock and data outputs as a parallel-to-serial converter.

Features of the Scan Manager

- Drives all the I/O scan chains for HPS I/O banks
- Allows the HPS to access the FPGA JTAG TAP controller

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

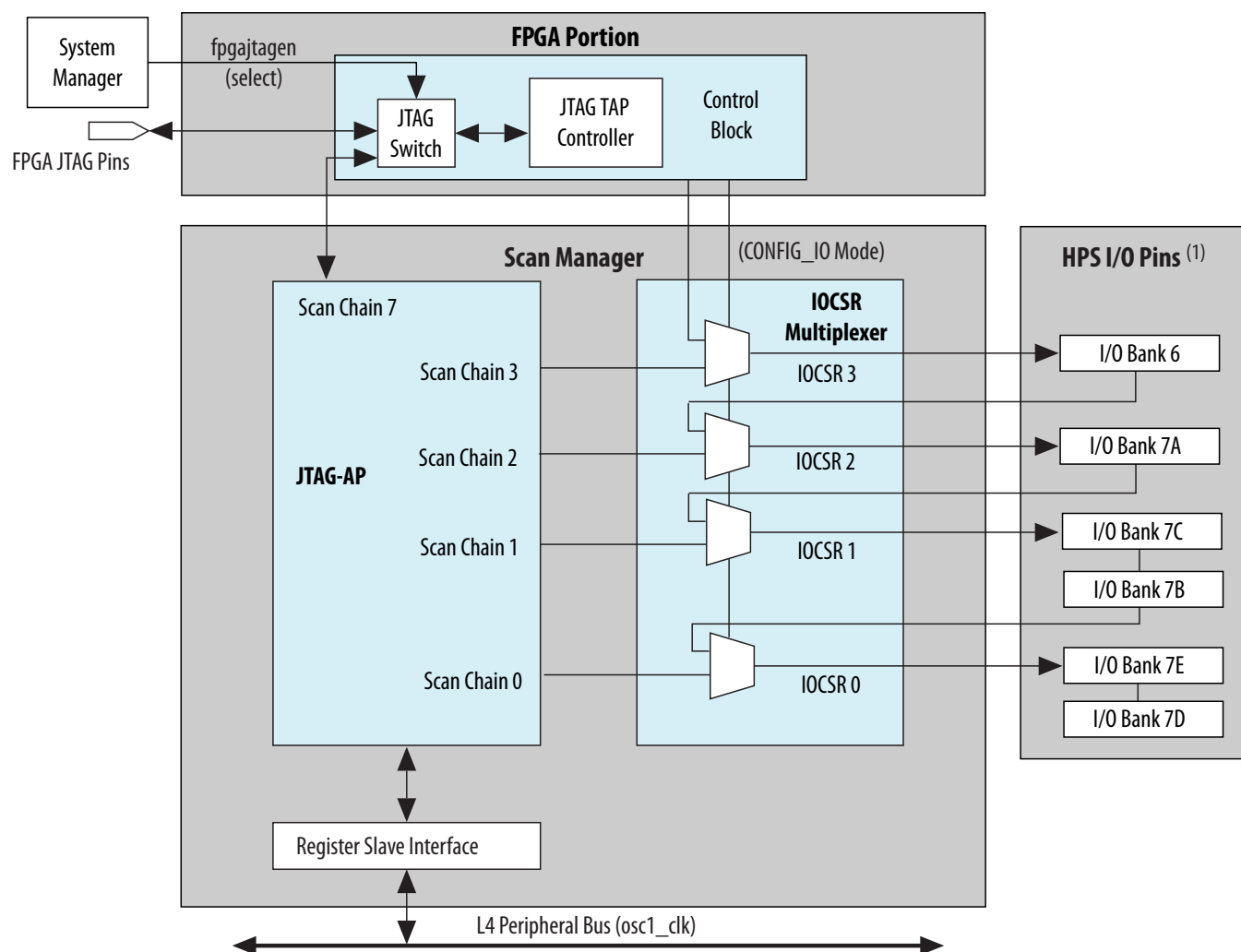
ISO
9001:2015
Registered

ALTERA
now part of Intel

Scan Manager Block Diagram and System Integration

Figure 7-1: Scan Manager Block Diagram

(1) Not all devices contain all the banks depicted.



The processor accesses the scan manager through the register slave interface connected to the level 4 (L4) peripheral bus.

Arm JTAG-AP Signal Use in the Scan Manager

The following table describes how the Arm JTAG-AP signals are connected in the scan manager. These signals are internal to the scan manager, are provided here for reference only, and are not shown in the preceding figure. The signal, register, and field names listed in the table match the names used in the *Arm Debug Interface v5 Architecture Specification*.

Signal	Direction	Implementation
SRSTCONNECTED[7:0]	Input	Tied to 0. The read-only SRSTCONNECTED field in the CSW register always reads as 0.
PORTCONNECTED[7:0]	Input	Tied to 0x8F, which connects only ports 0-3 and 7. The read-only PORTCONNECTED field in the CSW register reads as 1 when the PORTSEL register is written with a value that enables one of the connected ports, and reads as 0 otherwise.
PORTENABLED[7:0]	Input	Tied to 0x8F, so all connected ports are always considered powered on. The Arm JTAG AP PSTA register is not supported. Software does not need to monitor the status of ports 0-3 because they are always on. For port 7, software can read the mode field of the stat register in the FPGA manager to determine the FPGA power status.
nSRSTOUT[7:0]	Output	Not connected. Writing to the SRST_OUT field of the CSW register has no effect.
nTRST*[7:0]	Output	nTRST*[7] is connected to the FPGA JTAG TAP controller and nTRST*[6:0] are not connected. Writing to the TRST_OUT field of the CSW register (the trst bit of the stat register in the scan manager) has an effect only when port 7 is enabled by software. For details, refer to “Communicating with the JTAG TAP Controller”.

Related Information

- Information about configuring the scan manager's stat register
- [Communicating with the JTAG TAP Controller](#) on page 7-6

Arm JTAG-AP Scan Chains

The Arm JTAG-AP supports up to eight scan chains. The scan manager uses only scan chains 0, 1, 2, 3, and 7.

Scan chain 7 of the JTAG-AP connects to FPGA JTAG TAP controller. When the system manager undergoes a cold reset, this connection is disabled and the FPGA JTAG pins are connected to the FPGA JTAG TAP controller. You can configure the system manager to enable the connection, which allows software running on the HPS to communicate with the FPGA JTAG TAP controller. In this case, software can send JTAG commands (such as the SHIFT_EDERROR_REG JTAG instruction) to the FPGA JTAG and get responses to determine details about CRC errors detected by the control block when the FPGA fabric is in user mode. Through the FPGA manager, software can determine that a CRC error was detected. For more

information about the TAP controller, refer to the *Communicating with the JTAG TAP Controller* section of this chapter.

Scan chains 0 to 3 of the JTAG-AP connect to the configuration information in the HPS I/O scan chain banks through the I/O configuration shift register (IOCSR) multiplexer. For more information, refer to the *Configuring HPS I/O Scan Chains* section of this chapter.

Note: The I/O scan chains do not use the JTAG protocol. The scan manager uses the JTAG-AP as a parallel-to-serial converter for the I/O scan chains. The I/O scan chains are connected only to the serial output data (τ DI JTAG signal) and serial clock (τ CK JTAG signal).

The HPS I/O pins are divided into six banks. Each I/O bank is either a vertical (VIO) or horizontal (HIO) I/O, based on its location on the die.

Table 7-1: Bank Usage of IOCSR Scan Chains

The following table shows the mapping of the IOCSR scan chains to the I/O banks.

IOCSR Scan Chain	Bank Type	HPS I/O Bank	Usage
0	VIO	I/O bank 7D and I/O bank 7E	EMAC
1	VIO	I/O bank 7B and I/O bank 7C	SD/MMC, NAND, and quad SPI
2	VIO	I/O bank 7A	Trace, SPI, UART, and I ² C
3	HIO	I/O bank 6	SDRAM DDR

When the FPGA JTAG TAP controller is in `CONFIG_I0` mode, the controller can override the scan manager JTAG-AP and configure the HPS I/O pins. For more information, refer to the *Configuring HPS I/O Scan Chains* section of this chapter.

Note: `CONFIG_I0` mode is commonly used to configure the I/O pin properties prior to performing boundary scan testing.

Note: The HPS JTAG pins and the following HPS I/O pins do not support boundary scan tests (BST):

- DDR SDRAM
- OSC1/2
- Warm/Cold reset

To perform boundary scan testing on HPS I/O pins, use the FPGA JTAG.

Related Information

- [Configuring HPS I/O Scan Chains](#) on page 7-5
- [Communicating with the JTAG TAP Controller](#) on page 7-6

- [Arria V Device Handbook Volume 1: Device Interfaces and Integration](#)
For more information about boundary scan tests, refer to the "JTAG Boundary-Scan Testing in Arria V Devices" chapter.

Functional Description of the Scan Manager

The scan manager serves the following purposes:

- Configuring HPS I/O scan chains
- Communicating with the JTAG TAP controller

Configuring HPS I/O Scan Chains

The HPS I/O pins are configured through a series of scan chains.

I/O pin configuration involves such steps as setting the I/O standard and drive strength for each I/O bank. After a cold reset, all the I/O scan chains in the HPS must be configured prior to being used to communicate with external devices.

Software uses the scan manager to write configuration data to the scan chains. Separate I/O configuration data files for FPGA and HPS are generated by the Quartus® Prime software when the configuration image for the FPGA portion of the system-on-a-chip (SoC) device is assembled. The HPS configuration data is written to the scan manager by software.

Before configuring a specific I/O bank, the corresponding scan chain must be enabled by writing to the bits in the en register. The scan manager must not be active during this process. Software reads the active bit of the stat register to determine the scan manager state.

Alternatively, when the FPGA JTAG TAP controller receives the CONFIG_I0 JTAG instruction, the control block enters CONFIG_I0 mode. When the control block is in CONFIG_I0 mode, the controller can override the scan manager JTAG-AP and configure the HPS I/O pins. The CONFIG_I0 instruction configures all configurable I/O pins in the SoC device including the FPGA I/O pins and the HPS I/O pins. The FPGA and HPS portions of the device must both be powered on to execute the CONFIG_I0 instruction. External logic connected to the FPGA JTAG pins sends the CONFIG_I0 instruction, which provides I/O configuration data for all FPGA and HPS I/O pins. While CONFIG_I0 mode is active, the HPS is held in cold reset to prevent software from potentially interfering with the I/O configuration.

Related Information

- Information about configuring the scan manager's stat register
- Information about configuring the scan manager's en register
- [Scan Manager Address Map and Register Definitions](#) on page 7-8
- [Arria V Device Handbook Volume 1: Device Interfaces and Integration](#)
For more information about boundary scan tests, refer to the "JTAG Boundary-Scan Testing in Arria V Devices" chapter.
- [System Manager](#) on page 6-1
The HPS I/O pins need to be frozen before configuring them. For more information, refer to the *System Manager* chapter.

Communicating with the JTAG TAP Controller

After the system manager undergoes a cold reset, access to the JTAG TAP controller in the FPGA control block is through the dedicated FPGA JTAG I/O pins. If necessary, you can configure your system to use the scan manager to provide the HPS processor access to the JTAG TAP controller, instead. This feature allows the processor to send JTAG instructions to the FPGA portion of the device.

To connect scan chain 7 between the scan manager and the FPGA JTAG TAP controller, the following features must be enabled:

- The scan chain for the FPGA JTAG TAP controller—To enable scan chain 7, set the `fpga_jtag` field of the `en` register in the scan manager. For more information, refer to *"Scan Manager Address Map and Register Definitions"*.
- The FPGA JTAG logic source select—This source select determines whether the scan manager or the dedicated FPGA JTAG pins are connected to the FPGA JTAG TAP controller in the FPGA portion of the device. On system manager cold reset, the dedicated FPGA JTAG pins are selected. The source select is configured through the `fpga_jtag_en` bit of the `ctrl` register in the `scanmgrgrp` group of the system manager. The FPGA JTAG pins and scan manager connection to the TAP controller must both be inactive when switching between them. The mechanism to ensure both are inactive is user-defined.

Note: Before connecting or disconnecting the scan chain between the scan manager and the FPGA JTAG TAP controller, ensure that both the FPGA JTAG `TCK` and scan manager `TCK` signals are de-asserted. Altera recommends resetting the FPGA JTAG TAP controller using the scan manager's `nTRST` signal after the scan manager is connected to the controller.

Related Information

- Information about configuring the scan manager's `en` register
 - [Scan Manager Address Map and Register Definitions](#) on page 7-8
 - [System Manager](#) on page 6-1
- For information about the system manager, including details about configuring the `ctrl` register, refer to the *System Manager* chapter.

JTAG-AP FIFO Buffer Access and Byte Command Protocol

The JTAG-AP contains FIFO buffers for byte commands and responses. The buffers are accessed through the `fifosinglebyte`, `fifodoublebyte`, `fifotriplebyte`, and `fifoquadbyte` registers. The JTAG-AP stalls processor access to the registers when the buffer does not contain enough data for read access, or when the buffer does not contain enough free space to accept data for write access.

Note: Software should read the `rfifocnt` and `wfifocnt` fields of the `stat` register to determine the buffer status before performing the access to avoid being stalled by the JTAG-AP.

JTAG-AP scan chains 0, 1, 2 and 3 are write-only ports connected to the HPS IOCSRs and JTAG-AP scan chain 7 is a read-write port connected to the FPGA JTAG TAP controller. The processor can send data to scan chains 0-3, and send and receive data from scan chain 7 by accessing the command and response FIFO buffers in the JTAG-AP.

Note: Attempting to access data at invalid or non-aligned offsets can produce unpredictable results that require a reset to recover.

The JTAG commands and TDI data must be sent to the JTAG-AP using an encoded byte protocol. Similarly, the TDO data received from JTAG-AP is encoded. All commands are 8 bits wide in the byte command protocol.

Table 7-2: JTAG-AP Byte Command Protocol

Bits of the Command Byte								Opcode
7	6	5	4	3	2	1	0	
0	Opcode Payload							TMS
1	0	0	Opcode Payload					TDI_TDO
1	0	1	X	X	X	X	X	Reserved
1	1	0	X	X	X	X	X	Reserved
1	1	1	X	X	X	X	X	Reserved

Related Information

- Information about configuring the scan manager's fifosinglebyte register
- Information about configuring the scan manager's fifodoublebyte register
- Information about configuring the scan manager's fifotriplebyte register
- Information about configuring the scan manager's fifoquadbyte register
- Information about configuring the scan manager's stat register
- [Scan Manager Address Map and Register Definitions](#) on page 7-8

Clocks

The scan manager is connected to the `spi_m_clk` clock generated by the clock manager.

The scan manager generates two clocks. One clock routes to the control block of the FPGA portion of the SoC device with a frequency of $\text{spi\_m\_clk} / 6$ and runs at a maximum of 33 MHz. The other clock routes to the HPS I/O scan chains with a frequency of $\text{spi\_m\_clk} / 2$ and runs at a maximum frequency of 100 MHz.

Note: The `spi_m_clk` can potentially run faster than the scan manager supports so that SPI masters can support 60 Mbps rates. When the SPI master is running faster than what is supported by the scan manager, the scan manager cannot be used and must be held in reset.

Related Information

[Clock Manager](#) on page 3-1

For more information, including minimum and maximum clock frequencies, refer to the *Clock Manager* chapter.

Resets

The reset manager provides the `scan_manager_rst_n` reset signal to the scan manager for both cold and warm resets.

Because glitches can happen on the output clocks during a warm reset, the scan manager temporarily stops generation of the JTAG-AP and I/O configuration clocks. This action ensures that a warm reset does not cause output clock glitches.

Before asserting warm reset, the reset manager sends a request to the scan manager. The scan manager stops the output clock generation and acknowledges the reset manager. The reset manager then issues the warm reset. To enable this warm reset handshake, configure the `scanmgrhsen` bit of the reset manager `ctrl` register.

Related Information

- [Reset Manager](#) on page 4-1
For more information about reset handshaking, refer to the *Reset Manager* chapter.
- [System Manager](#) on page 6-1
For information about the system manager, including details about configuring the `ctrl` register, refer to the *System Manager* chapter.

Scan Manager Address Map and Register Definitions

This section lists the scan manager register address map and describes the registers.

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
The base addresses of all modules are also listed in the *Introduction to the Hard Processor* chapter.
- [Arria V Address Map and Register Definitions](#)
Web-based address map and register definitions

JTAG-AP Register Name Cross Reference Table

To clarify how Altera uses the JTAG-AP, the Arm registers are renamed in the SoC device. The following table cross references the Arm and Altera register names.

Table 7-3: JTAG-AP Register Names

Altera Register Name	Arm Register Name
stat	CSW (control/status word)
en	PSEL
fifosinglebyte	BWFIF01 for writes, BRFIF01 for reads
fifodoublebyte	BWFIF02 for writes, BRFIF02 for reads
fifotriplebyte	BWFIF03 for writes, BRFIF03 for reads
fifoquadbyte	BWFIF04 for writes, BRFIF04 for reads

2021.05.07

av_5v4



Subscribe



Send Feedback

The components of the hard processor system (HPS) communicate with one another, and with other portions of the SoC device, through the system interconnect. The system interconnect consists of the following blocks:

- The main level 3 (L3) interconnect
- The level 4 (L4) buses

The system interconnect is implemented with the Arm CoreLink[®] Network Interconnect (NIC-301). The NIC-301 provides a foundation for a high-performance HPS system interconnect based on the Arm Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI), Advanced High-Performance Bus (AHB), and Advanced Peripheral Bus (APB) protocols. The system interconnect implements a multilayer, nonblocking architecture that supports multiple simultaneous transactions between masters and slaves, including the Cortex-A9 microprocessor unit (MPU) subsystem. The system interconnect provides five independent L4 buses to access control and status registers (CSRs) of peripherals, managers, and memory controllers.

Features of the System Interconnect

The system interconnect supports high-throughput peripheral devices. The system interconnect has the following characteristics:

- Main internal data width of 64 bits
- Programmable master priority with single-cycle arbitration
- Full pipelining to prevent master stalls
- Programmable control for FIFO buffer transaction release
- Arm TrustZone[®] compliant, with additional security features configurable per master
- Multiple independent L4 buses

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

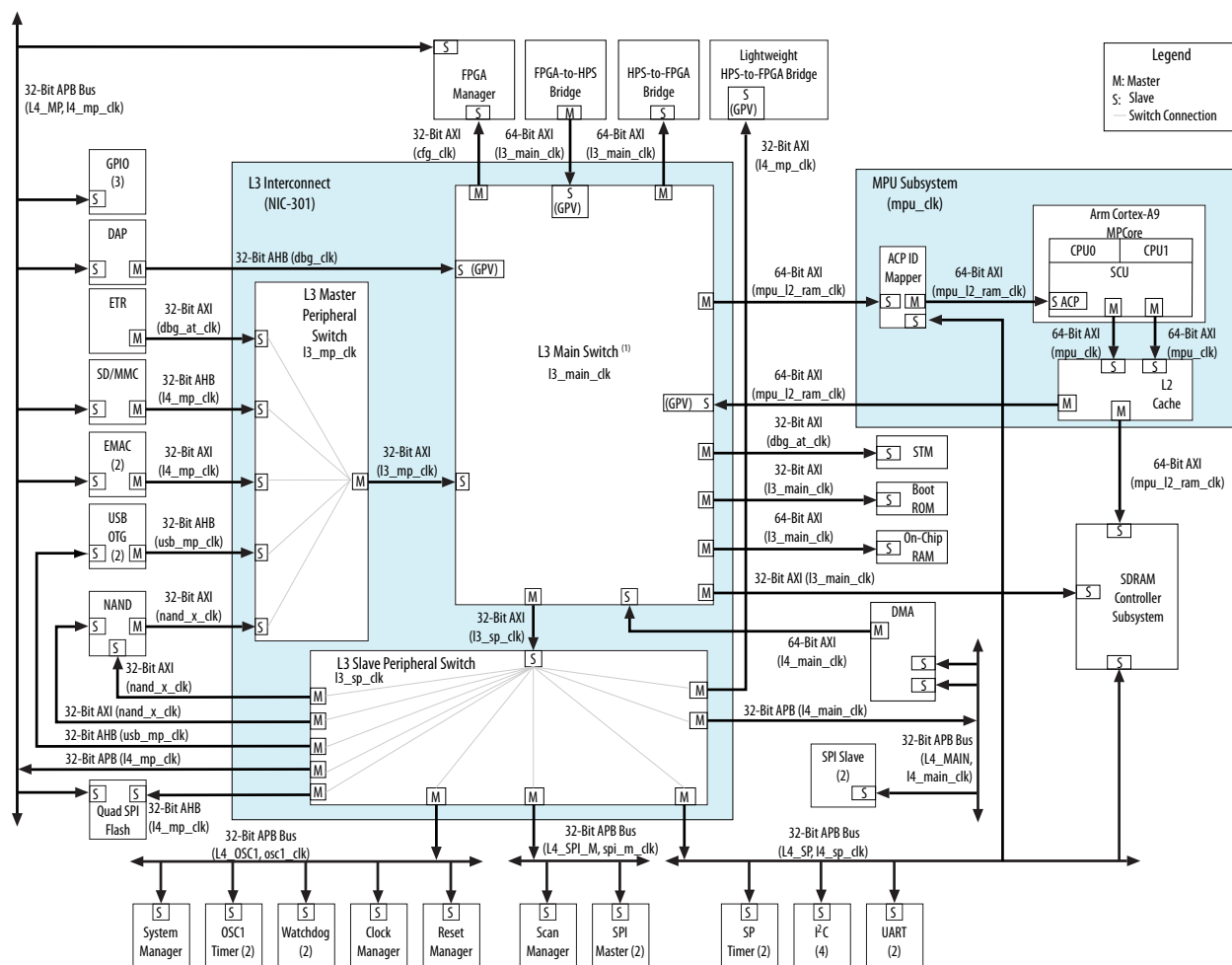
ISO
9001:2015
Registered

ALTERA
now part of Intel

System Interconnect Block Diagram and System Integration

Interconnect Block Diagram

Figure 8-1: L3 Interconnect and L4 Buses



Note: System interconnect slaves are available for connection from peripheral masters. System interconnect masters connect to peripheral slaves. This terminology is the reverse of conventional terminology used in Platform Designer (Standard).

Related Information

- **Master to Slave Connectivity Matrix** on page 8-4
- **Main Connectivity Matrix** on page 8-3

System Interconnect Architecture

The L3 interconnect is a partially-connected switch fabric. Not all masters can access all slaves.

Internally, the L3 interconnect is partitioned into the following subswitches:

- L3 interconnect
 - Interconnect used to transfer high-throughput 64-bit data
 - Operates at up to half the MPU main clock frequency
 - Provides masters with low-latency connectivity to AXI bridges, on-chip memories, SDRAM, and FPGA manager
- L3 master peripheral switch
 - Used to connect memory-mastering peripherals to the interconnect
 - 32-bit data width
 - Operates at up to half the interconnect clock frequency
- L3 slave peripheral switch
 - Used to provide access to level 3 and 4 slave interfaces for masters of the master peripheral and interconnects
 - 32-bit data width
 - Five independent L4 buses

The L3 master and slave peripheral switches are fully-connected crossbars. The L3 interconnect is a partially-connected crossbar. The following table shows the connectivity matrix of all the master and slave interfaces of the L3 interconnect.

Main Connectivity Matrix

The L3 master and slave peripheral switches are fully-connected crossbars. The L3 interconnect is a partially-connected crossbar. The following table shows the connectivity matrix of all the master and slave interfaces of the L3 interconnect.

⁽¹⁴⁾ For details of the masters and slaves connected to the L3 master peripheral switch and L3 master peripheral switch, refer to "Interconnect Block Diagram".

Figure 8-2: Main Connectivity Matrix

Masters	Slaves							
	L3 Slave Peripheral Switch (1)	FPGA Manager	HPS-to-FPGA Bridge	ACP ID Mapper Data	STM	Boot ROM	On-Chip RAM	SDRAM Controller Subsystem L3 Data
L3 Master Peripheral Switch (1)			✓	✓			✓	✓
L2 Cache Master 0	✓	✓	✓		✓	✓	✓	
FPGA-to-HPS Bridge	✓			✓	✓		✓	✓
DMA	✓	✓	✓	✓	✓		✓	✓
DAP	✓	✓	✓	✓			✓	✓

Related Information

[Interconnect Block Diagram](#) on page 8-2

Functional Description of the Interconnect

Master to Slave Connectivity Matrix

The interconnect is a partially-connected crossbar. The following table shows the connectivity matrix of all the master and slave interfaces of the interconnect.

Figure 8-3: Interconnect Connectivity Matrix

Masters	Slaves																
	L4 SP Bus Slaves	L4 MP Bus Slaves	L4 OSC1 Bus Slaves	L4 MAIN Bus Slaves	L4 SPIM Bus Slaves	Lightweight HPS-to-FPGA Bridge	USB OTG 0/1 CSR	NAND CSR	NAND Command and Data	Quad SPI Flash Data	FPGA Manager	HPS-to-FPGA Bridge	ACP ID Mapper Data	STM	Boot ROM	On-Chip RAM	SDRAM Controller Subsystem L3 Data
L2 Cache Master 0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	
FPGA-to-HPS Bridge	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓		✓	✓
DMA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
EMAC 0/1												✓	✓			✓	✓
USB OTG 0/1												✓	✓			✓	✓
NAND												✓	✓			✓	✓
SD/MMC												✓	✓			✓	✓
ETR												✓	✓			✓	✓
DAP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓

System Interconnect Address Spaces

The system interconnect supports multiple address spaces.

Each address space uses some or all of the 4 GB address range. The address spaces overlap. Depending on the configuration, different address spaces are visible in different regions for each master.

The following address spaces are available:

- The L3 address space
- The MPU address space
- The SDRAM address space

Available Address Maps

The following figure shows the default system interconnect address maps for all masters. The figure is not to scale.

Figure 8-4: Address Maps for System Interconnect Masters

	DMA	Master Peripherals (6)	DAP	FPGA-to-HPS Bridge	MPU
0xFFFFFFFF	On-Chip RAM	On-Chip RAM	On-Chip RAM	On-Chip RAM	On-Chip RAM
0xFFFF0000					SCU and L2 Registers (1)
0xFFFE0000					Boot ROM
0xFFD00000	Peripherals and L3 GPV		Peripherals and L3 GPV	Peripherals and L3 GPV	Peripherals and L3 GPV
0xFF400000	LW H-to-F (2)		LW H-to-F (2)	LW H-to-F (2)	LW H-to-F (2)
0xFF200000					
0xFF000000	DAP		DAP	DAP	DAP
0xFF000000	STM			STM	STM
0xFC000000					
0xC0000000	H-to-F (3)	H-to-F (3)	H-to-F (3)		H-to-F (3)
0xC0000000	ACP	ACP	ACP	ACP	
0x80000000					
0x80000000	SDRAM	SDRAM	SDRAM	SDRAM	SDRAM (4)
0x00100000					
0x00010000					
0x00000000	SDRAM (5)	SDRAM (5)	SDRAM (5)	SDRAM (5)	Boot ROM

Notes on Address Maps

- (1) Transactions on these addresses are directly decoded by the SCU and L2 cache.
- (2) This region can be configured to access slaves on the lightweight HPS-to-FPGA bridge, by using the remap register.
- (3) This region can be configured to access slaves on the HPS-to-FPGA bridge, by using the remap register.
- (4) The MPU accesses SDRAM through a dedicated port.
- (5) This region can be configured to access on-chip RAM, by using the remap register.

(6) The following peripherals can master the interconnect:

- Ethernet MACs
- USB-2 OTG controllers
- NAND controllers
- ETR
- SD/MMC controller

For the MPU master, either the boot ROM or on-chip RAM maps to address 0x0 and obscures the lowest 64 KB of SDRAM.

At boot time, the MPU does not have access to the SDRAM address space from 0x00010000 to 0x00100000. This is because the MPU's SDRAM access is controlled by the MPU L2 filter registers, which only have a granularity of 1 MB. After booting completes, the MPU can change address filtering to use the lowest 1 MB of SDRAM.

For non-MPU masters, either the on-chip RAM or the SDRAM maps to address 0x0. When mapped to address 0x0, the on-chip RAM obscures the lowest 64 KB of SDRAM for non-MPU masters.

Related Information

- [Cortex-A9 MPU Subsystem Register Implementation](#)
More information about MPU register implementation, including the SCU and L2 cache registers
- [The SDRAM Region](#)
More information about using L2 address filtering to control SDRAM access
- [Address Remapping](#) on page 8-11
Information about controlling the system interconnect memory maps

L3 Address Space

The L3 address space is 4 GB and applies to all L3 masters except the MPU subsystem.

The L3 address space configurations contain the regions shown in the following table:

Table 8-1: L3 Address Space Regions

Description	Condition	Base Address	End Address	Size
SDRAM window (without on-chip RAM)	When <code>remap.nonmpuzero</code> ⁽¹⁵⁾ is clear	0x00000000	0xBFFFFFFF	3 GB
On-chip RAM	When <code>remap.nonmpuzero</code> ⁽¹⁵⁾ is set	0x00000000	0x0000FFFF	64 KB
SDRAM window (with on-chip RAM)	When <code>remap.nonmpuzero</code> ⁽¹⁵⁾ is set	0x00010000	0xBFFFFFFF	3145664 KB = 3 GB - 64 KB
ACP window	Always visible	0x80000000	0xBFFFFFFF	1 GB
HPS-to-FPGA	When <code>remap.hps2fpga</code> ⁽¹⁵⁾ is set. Not visible to FPGA-to-HPS bridge.	0xC0000000	0xFBFFFFFFF	960 MB
System trace macrocell	Always visible to DMA and FPGA-to-HPS	0xFC000000	0xFEFFFFFFF	48 KB

⁽¹⁵⁾ For details about the remap register, refer to "Bit Fields for Modifying the Memory Map"

Description	Condition	Base Address	End Address	Size
Debug access port	Not visible to master peripherals. Always visible to other masters.	0xFF000000	0xFF1FFFFFFF	2 MB
Lightweight HPS-to-FPGA	Not visible to master peripherals. Visible to other masters when <code>remap.hps2fpga⁽¹⁵⁾</code> is set.	0xFF200000	0xFF3FFFFFFF	2 MB
Peripherals	Not visible to master peripherals. Always visible to other masters.	0xFF400000	0xFFFCFFFF	12096 KB
On-chip RAM	Always visible	0xFFFF0000	0xFFFFFFFF	64 KB

The boot ROM and internal MPU registers (SCU and L2) are not accessible to L3 masters.

SDRAM Window Region

The SDRAM window region is 3 GB and provides access to the bottom 3 GB of the SDRAM address space. Any L3 master can access a cache-coherent view of SDRAM by performing a cacheable access through the ACP.

On-Chip RAM Region

The system interconnect remap register, in the `l3regs` group, determines if the 64 KB starting at address 0x0 is mapped to the on-chip RAM or the SDRAM. The SDRAM is mapped to address 0x0 on reset.

ACP Window Region

The ACP window region is 1 GB and provides access to a configurable gigabyte-aligned region of the MPU address space. Registers in the ACP ID mapper control which gigabyte-aligned region of the MPU address space is accessed by the ACP window region. The ACP window region is used by L3 masters to perform coherent accesses into the MPU address space. For more information about the ACP ID mapper, refer to the *Cortex-A9 Microprocessor Unit Subsystem* chapter.

HPS-to-FPGA Slaves Region

The HPS-to-FPGA slaves region provides access to 960 MB of slaves in the FPGA fabric through the HPS-to-FPGA bridge.

Lightweight HPS-to-FPGA Slaves Region

The lightweight HPS-to-FPGA slaves provide access to slaves in the FPGA fabric through the lightweight HPS-to-FPGA bridge.

Peripherals Region

The peripherals region includes slaves connected to the L3 interconnect and L4 buses.

On-Chip RAM Region

The on-chip RAM is always mapped (independent of the boot region contents).

Related Information

- [Bit Fields for Modifying the Memory Map](#) on page 8-12
- [Cortex-A9 Microprocessor Unit Subsystem](#)
For general information about the MPU subsystem, refer to the *Cortex-A9 Microprocessor Unit Subsystem* chapter.

MPU Address Space

The MPU address space is 4 GB and applies to MPU masters.

Addresses generated by the MPU are decoded in three ways:

- By default, MPU accesses to locations between 0x100000 (1 MB) to 0xC0000000 (3 GB) are made to the SDRAM controller.
- Addresses in the SCU and L2 register region (0xFFFE0000 to 0xFFFF0000) are the SCU and L2 bus.
- Accesses to all other locations are made to the L3 interconnect.

The MPU L2 cache controller contains a master connected to the system interconnect and a master connected to the SDRAM.

The MPU address space contains the following regions:

Table 8-2: MPU Default Address Space Regions

Description	Condition	Base Address	End Address	Size
Boot ROM	Always visible	0x00000000	0x000FFFFFFF	1 MB
SDRAM window	Always visible	0x00100000	0xBFFFFFFF	3047 MB (3 GB – 1 MB)
HPS-to-FPGA	When <code>remap.hps2fpga</code> ⁽¹⁶⁾ is set.	0xC0000000	0xFBFFFFFFF	348 KB
System trace macrocell	Always visible	0xFC000000	0xFFEFFFFFFF	48 KB
Debug access port	Always visible	0xFF000000	0xFF1FFFFFFF	2 MB
Lightweight HPS-to-FPGA	Visible when <code>remap.hps2fpga</code> ⁽¹⁶⁾ is set.	0xFF200000	0xFF3FFFFFFF	2 MB
Peripherals	Always visible	0xFF400000	0xFFFCFFFF	12096 KB
Boot ROM	Always visible	0xFFFD0000	0xFFFE0000	112 KB
SCU and L2 registers	Always visible	0xFFFE0000	0xFFFF0000	16 KB

⁽¹⁶⁾ For details about the remap register, refer to "Bit Fields for Modifying the Memory Map"



Description	Condition	Base Address	End Address	Size
On-chip RAM	Always visible	0xFFFF0000	0xFFFFFFFF	64 KB

Boot Region

The boot region is 1 MB, based at address 0x0. The boot region is visible to the MPU only when the L2 address filter start register is set to 0x100000. The L3 remap control register determines if the boot region is mapped to the on-chip RAM or the boot ROM.

The boot region is mapped to the boot ROM on reset. Only the lowest 64 KB of the boot region are legal addresses because the on-chip RAM and boot ROM are only 64 KB.

When the L2 address filter start register is set to 0, SDRAM obscures access to the boot region. This technique can be used to gain access to the lowest SDRAM addresses after booting completes.

SDRAM Window Region

The SDRAM window region provides access to a large, configurable portion of the 4 GB SDRAM address space. The address filtering start and end registers in the L2 cache controller define the SDRAM window boundaries.

The boundaries are megabyte-aligned. Addresses within the boundaries map to the SDRAM controller, which queues the read and write transactions for execution. Addresses that fall outside the boundaries route to the system interconnect, to access peripherals, bridges, and other HPS resources.

Addresses in the SDRAM window match addresses in the SDRAM address space. Thus, the lowest 1 MB of the SDRAM is not visible to the MPU unless the L2 address filter start register is set to 0.

HPS-to-FPGA Slaves Region

The HPS-to-FPGA slaves region provides access to slaves in the FPGA fabric through the HPS-to-FPGA bridge. Software can move the top of the SDRAM window by writing to the L2 address filter end register. If higher addresses are made available in the SDRAM window, part of the FPGA slaves region might be inaccessible to the MPU.

Lightweight HPS-to-FPGA Slaves Region

The lightweight FPGA slaves provide access to slaves in the FPGA fabric through the lightweight HPS-to-FPGA bridge.

Peripherals Region

The peripherals region is near the top of the address space. The peripheral region includes slaves connected to the L3 interconnect and L4 buses.

Boot ROM Region

The boot ROM is always mapped near the top of the address space, independent of the boot region contents.

SCU and L2 Registers Region

The SCU and L2 registers region provides access to internally-decoded MPU registers (SCU and L2).

On-Chip RAM Region

The on-chip RAM is always mapped near the top of the address space, independent of the boot region contents.

Related Information

- [HPS Address Spaces](#) on page 2-15
Figure showing the SDRAM window boundaries at reset
- [The SDRAM Region](#)
Information about L2 cache filtering, including the address filter start and address filter end registers
- [Bit Fields for Modifying the Memory Map](#) on page 8-12

SDRAM Address Space

The SDRAM address space is 4 GB. Depending on register settings, portions of it are visible to all masters.

Related Information

[HPS Address Spaces](#) on page 2-15
Figure showing the SDRAM window boundaries at reset

Address Remapping

The system interconnect supports address remapping through the remap register in the l3regs group. Remapping allows software to control which memory device (SDRAM, on-chip RAM, or boot ROM) is accessible at address 0x0 and the accessibility of the HPS-to-FPGA and lightweight HPS-to-FPGA bridges. The remap register is one of the NIC-301 Global Programmers View (GPV) registers. The following L3 masters can manipulate remap, because it maps into their address space:

- MPU
- FPGA-to-HPS bridge
- DAP

The remapping bits in the remap register are not mutually exclusive. The lowest order remap bit has higher priority when multiple slaves are remapped to the same address. Each bit allows different combinations of address maps to be formed. There is only one remapping register available in the GPV, so modifying the remap register affects all memory maps of all the masters of the system interconnect.

The effects of the remap bits can be categorized in the following groups:

- MPU master interface
 - L2 cache master 0 interface
- Non-MPU master interfaces
 - DMA master interface
 - Master peripheral interfaces
 - Debug Access Port (DAP) master interface
 - FPGA-to-HPS bridge master interface

Note: L2 filter registers in the MPU subsystem, not the interconnect, allow the SDRAM to be remapped to address 0x0 for the MPU.

Related Information

- **The SDRAM Region**
Information about L2 cache filtering, including the address filter start and address filter end registers
- **Cortex-A9 Microprocessor Unit Subsystem**
For general information about the MPU subsystem, refer to the *Cortex-A9 Microprocessor Unit Subsystem* chapter.
- **HPS Peripheral Master Input IDs**
For information about virtual ID mapping in the ACP ID mapper, refer to "HPS Peripheral Master Input IDs" in the *Cortex-A9 Microprocessor Unit Subsystem* chapter.

Bit Fields for Modifying the Memory Map**Table 8-3: remap Bit Fields**

Bit Name	Bit Offset	Description
mpuzero	0	Value Meaning 0 The boot ROM maps to address 0x0 for the MPU L3 master 1 The on-chip RAM maps to address 0x0 for the MPU L3 master This bit has no effect on non-MPU masters. Note: Regardless of this setting, the boot ROM also always maps to address 0xFFFF0000 and the on-chip RAM also always maps to address 0xFFFD0000 for the MPU L3 master.
nonmpuzero	1	Value Meaning 0 The SDRAM maps to address 0x0 for the non-MPU L3 masters 1 The on-chip RAM maps to address 0x0 for the non-MPU masters This bit has no effect on the MPU L3 master. Note that regardless of this setting, the on-chip RAM also always maps to address 0xFFFD0000 for the non-MPU L3 masters.
Reserved	2	Must always be 0.
hps2fpga	3	Value Meaning 0 Accesses to the associated address range return an AXI decode error to the master 1 The HPS-to-FPGA bridge slave port is visible to the L3 masters

Bit Name	Bit Offset	Description						
lwhp2fpga	4	<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0</td><td>Accesses to the associated address range return an AXI decode error to the master</td></tr><tr><td>1</td><td>The lightweight HPS-to-FPGA bridge slave port is visible to the L3 masters</td></tr></table>	Value	Meaning	0	Accesses to the associated address range return an AXI decode error to the master	1	The lightweight HPS-to-FPGA bridge slave port is visible to the L3 masters
Value	Meaning							
0	Accesses to the associated address range return an AXI decode error to the master							
1	The lightweight HPS-to-FPGA bridge slave port is visible to the L3 masters							
Reserved	31:5	Must always be 0.						

Master Caching and Buffering Overrides

Some of the peripheral masters connected to the system interconnect do not have the ability to drive the caching and buffering signals of their interfaces. The system manager provides registers so that you can enable cacheable and bufferable transactions for these masters. The system manager drives the caching and buffering signals of the following masters:

Master Peripheral	System Manager Register Group	Register
EMAC0 and EMAC1	emacgrp	l3master
USB OTG 0 and USB OTG 1	usbgrp	l3master
NAND flash	nandgrp	l3master
SD/MMC	sdmmcgrp	l3master

At reset time, the system manager drives the cache and buffering signals for these masters low. In other words, the masters listed do not support cacheable or bufferable accesses until you enable them after reset. There is no synchronization between the system manager and the system interconnect, so avoid changing these settings when any of the masters are active.

Related Information

[System Manager](#) on page 6-1

For more information about enabling or disabling these features, refer to the *System Manager* chapter.

Security

Slave Security

The interconnect enforces security through the slave settings. The slave settings are controlled by the address region control registers accessible through the GPV registers. Each L3 and L4 slave has its own security check and programmable security settings. After reset, every slave of the interconnect is set to a secure state (referred to as boot secure). The only accesses allowed to secure slaves are by secure masters.

The GPV can only be accessed by secure masters. The security state of the interconnect is not accessible through the GPV as the security registers are write-only. Any nonsecure accesses to the GPV receive a DECERR response, and no register access is provided. Updates to the security settings through the GPV do not take effect until all transactions to the affected slave have completed.

Master Security

Masters of the system interconnect are either secure, nonsecure, or the security is set on a per transaction basis. The DAP and Ethernet masters only perform secure accesses. The L2 cache master 0, FPGA-to-HPS-bridge, and DMA perform secure and nonsecure accesses on a per transaction basis. All other system interconnect masters perform nonsecure accesses.

Accesses to secure slaves by unsecure masters result in a response of DECERR and the transaction does not reach the slave.

All masters are secure at reset.

Related Information

[System Interconnect Master Properties](#) on page 8-15

Configuring the Quality of Service Logic

You can programmatically configure the QoS generator for each master through the QoS registers.

Note: Before accessing the registers for any QoS generator, you must ensure that the corresponding peripheral is not in reset. Otherwise, the register access results in a bus error, causing a CPU fault.

Cyclic Dependency Avoidance Schemes

The AXI protocol permits re-ordering of transactions. As a result, when routing concurrent multiple transactions from a single point of divergence to multiple slaves, the interconnect might need to enforce rules to prevent deadlock.

Each master of the interconnect is configured with one of three possible cyclic dependency avoidance schemes (CDAS). The same CDAS scheme is configured for both read and write transactions, but they operate independently.

[Single Slave](#) on page 8-14

[Single Slave Per ID](#) on page 8-15

[Single Active Slave](#) on page 8-15

Related Information

[System Interconnect Master Properties](#) on page 8-15

Contains descriptions of the CDAS implementation for the masters.

Single Slave

Single slave (SS) ensures the following conditions at a slave interface of a switch:

- All outstanding read transactions are to a single end destination.
- All outstanding write transactions are to a single end destination.

If a master issues another transaction to a different destination than the current destination for that transaction type (read or write), the network stalls the transactions until all the outstanding transactions of that type have completed.

Single Slave Per ID

Single slave per ID (SSPID) ensures the following conditions at a slave interface of a switch:

- All outstanding read transactions with the same ID go the same destination.
- All outstanding write transactions with the same ID go the same destination.

When a master issues a transaction, the following situations can occur:

- If the transaction has an ID that does not match any outstanding transactions, it passes the CDAS.
- If the transaction has an ID that matches the ID of an outstanding transaction, and the destinations also match, it passes the CDAS.
- If the transaction has an ID that matches the ID of an outstanding transaction, and the destinations do not match, the transaction fails the CDAS check and stalls.

Single Active Slave

Single active slave (SAS) is the same as the SSPID scheme, with an added check for write transactions. SAS ensures that a master cannot issue a new write address until all of the data from the previous write transaction has been sent.

System Interconnect Master Properties

The system interconnect connects to various slave interfaces through the L3 interconnect and L3 slave peripheral switch.

Table 8-4: System Interconnect Master Interfaces

TrustZone security:

- Secure: All transactions are marked TrustZone secure
- Nonsecure: All transactions are marked TrustZone non-secure
- Per transaction: Transactions can be marked TrustZone secure or TrustZone non-secure, depending on the state of the system interconnect master.

Issuance is based on the number of read, write, and total transactions.

The FIFO buffer depth for AXI is based on the AW, AR, R, W, and B channels. For AHB and APB, the depth is based on W, A, and D channels.

Master	Width	Clock	Switch	TrustZone Security	GPV Access	CDAS	Issuance	FIFO Buffer Depth ⁽¹⁾ ₇₎	Type
L2 cache M0	64	mpu_l2_ram_clk	L3 interconnect	Per Transaction	Yes	SSPID	7, 12, 19	2, 2, 2, 2, 2	AXI

⁽¹⁷⁾ Each channel has a dedicated FIFO buffer. This allows the channels to function as independent streams.

Master	Width	Clock	Switch	TrustZone Security	GPV Access	CDAS	Issuance	FIFO Buffer Depth ⁽¹⁾	Type
FPGA-to-HPS bridge	64	l3_main_clk	L3 interconnect	Per Transaction	Yes	SAS	16, 16, 32	2, 2, 6, 6, 2	AXI
DMA	64	l4_main_clk	L3 interconnect	Per Transaction	No	SSPID	8, 8, 8	2, 2, 2, 2, 2	AXI
EMAC 0/1	32	l4_main_clk	L3 master peripheral switch	Secure	No	SSPID	16, 16, 32	2, 2, 2, 2, 2	AXI
USB OTG 0/1	32	usb_mp_clk	L3 master peripheral switch	Nonsecure	No	SSPID	2, 2, 4	2, 2, 2	AHB
NAND	32	nand_x_clk	L3 master peripheral switch	Nonsecure	No	SSPID	1, 8, 9	2, 2, 2, 2, 2	AXI
SD/MMC	32	l4_mp_clk	L3 master peripheral switch	Nonsecure	No	SSPID	2, 2, 4	2, 2, 2	AHB
ETR	32	dbg_at_clk	L3 master peripheral switch	Per Transaction	No	SSPID	32, 1, 32	2, 2, 2, 2, 2	AXI
DAP	32	dbg_clk	L3 interconnect	Secure	Yes	SS	1, 1, 1	2, 2, 2	AHB

The AXI IDs of the HPS peripheral masters identify transactions from the HPS to soft logic over the HPS-to-FPGA bridge. Soft logic in the FPGA can monitor the AXI IDs to determine which master issued each transaction. For HPS peripheral AXI IDs, refer to "HPS Peripheral Master Input IDs".

Related Information

[HPS Peripheral Master Input IDs](#)

For a list of HPS peripheral AXI IDs, refer to the *Cortex-A9 Microprocessor Unit Subsystem* chapter.

⁽¹⁷⁾ Each channel has a dedicated FIFO buffer. This allows the channels to function as independent streams.

Interconnect Slave Properties

The interconnect connects to various slave interfaces through the L3 interconnect, L3 slave peripheral switch, and the five L4 peripheral buses. After reset, all slave interfaces are set to the secure state.

The interconnect provides FIFO buffers with clock crossing adapters. Refer to "FIFO Buffers and Clock Crossing" for details.

Table 8-5: Interconnect Slave Interfaces

Slave	I/F Width	Clock	Mastered By	Acceptance ⁽¹⁸⁾	Buffer Depth ⁽¹⁹⁾	Type
SDRAM subsystem CSR	32	l4_sp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
SP timer 0/1	32	l4_sp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
I ² C 0/1/2/3	32	l4_sp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
UART 0/1	32	l4_sp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
GPIO 0/1/2	32	l4_mp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
ACP ID mapper CSR	32	l4_mp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
FPGA manager CSR	32	l4_mp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
DAP CSR	32	l4_mp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
Quad SPI flash CSR	32	l4_mp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
SD/MMC CSR	32	l4_mp_clk	L4 SP bus master	1, 1, 1	2, 2, 2	APB
EMAC 0/1 CSR	32	l4_mp_clk	L4 MP bus master	1, 1, 1	2, 2, 2	APB
System manager	32	osc1_clk	L4 OSC1 bus master	1, 1, 1	2, 2, 2	APB

⁽¹⁸⁾ Acceptance is based on the number of read, write, and total transactions.

⁽¹⁹⁾ The FIFO buffer depth for AXI is based on the AW, AR, R, W, and B channels. For AHB and APB, the depth is based on W, A, and D channels.

Slave	I/F Width	Clock	Mastered By	Acceptance ⁽¹⁸⁾	Buffer Depth ⁽¹⁹⁾	Type
OSC1 timer 0/1	32	osc1_clk	L4 OSC1 bus master	1, 1, 1	2, 2, 2	APB
Watchdog 0/1	32	osc1_clk	L4 OSC1 bus master	1, 1, 1	2, 2, 2	APB
Clock manager	32	osc1_clk	L4 OSC1 bus master	1, 1, 1	2, 2, 2	APB
Reset manager	32	osc1_clk	L4 OSC1 bus master	1, 1, 1	2, 2, 2	APB
DMA secure CSR	32	l4_main_clk	L4 main bus master	1, 1, 1	2, 2, 2	APB
DMA nonsecure CSR	32	l4_main_clk	L4 main bus master	1, 1, 1	2, 2, 2	APB
SPI slave 0/1	32	l4_main_clk	L4 main bus master	1, 1, 1	2, 2, 2	APB
Scan manager	32	spi_m_clk	L4 main bus master	1, 1, 1	2, 2, 2	APB
SPI master 0/1	32	spi_m_clk	L4 main bus master	1, 1, 1	2, 2, 2	APB
Lightweight HPS-to-FPGA bridge	32	l4_main_clk	L3 slave peripheral switch	16, 16, 32	2, 2, 2, 2, 2	AXI
USB OTG 0/1	32	usb_mp_clk	L3 slave peripheral switch	1, 1, 1	2, 2, 2	AHB
NAND CSR	32	nand_x_clk	L3 slave peripheral switch	1, 1, 1	2, 2, 2	AXI
NAND command and data	32	nand_x_clk	L3 slave peripheral switch	1, 1, 1	2, 2, 2	AXI

⁽¹⁸⁾ Acceptance is based on the number of read, write, and total transactions.

⁽¹⁹⁾ The FIFO buffer depth for AXI is based on the AW, AR, R, W, and B channels. For AHB and APB, the depth is based on W, A, and D channels.

Slave	I/F Width	Clock	Mastered By	Acceptance ⁽¹⁸⁾	Buffer Depth ⁽¹⁹⁾	Type
Quad SPI flash data	32	l4_mp_clk	L3 slave peripheral switch	1, 1, 1	2, 2, 2	AHB
FPGA manager data	32	cfg_clk	L3 interconnect	1, 2, 3	2, 2, 2, 32, 2	AXI
HPS-to-FPGA bridge	64	l3_main_clk	L3 interconnect	16, 16, 32	2, 2, 6, 6, 2	AXI
ACP ID mapper data	64	mpu_l2_ram_clk	L3 interconnect	13, 5, 18	2, 2, 2, 2, 2	AXI
STM	32	dbg_at_clk	L3 interconnect	1, 2, 2	2, 2, 2, 2, 2	AXI
On-chip boot ROM	32	l3_main_clk	L3 interconnect	1, 1, 2	0, 0, 0, 0, 0	AXI
On-chip RAM	64	l3_main_clk	L3 interconnect	2, 2, 2	0, 0, 0, 8, 0	AXI
SDRAM subsystem L3 data	32	l3_main_clk	L3 interconnect	16, 16, 16	2, 2, 2, 2, 2	AXI

Related Information

[FIFO Buffers and Clock Crossing](#) on page 8-21

Upsizing Data Width Function

The upsizing function combines narrow transactions into wider transactions to increase the overall system bandwidth. Upsizing only packs data for read or write transactions that are cacheable. If the interconnect splits input-exclusive transactions into more than one output bus transaction, it removes the exclusive information from the multiple transactions it creates.

The upsizing function can expand the data width by the following ratios:

- 1:2
- 1:4

⁽¹⁸⁾ Acceptance is based on the number of read, write, and total transactions.

⁽¹⁹⁾ The FIFO buffer depth for AXI is based on the AW, AR, R, W, and B channels. For AHB and APB, the depth is based on W, A, and D channels.

If multiple responses from created transactions are combined into one response, then the following order of priority applies:

- DECERR is the highest priority
- SLVERR is the next highest priority
- OKAY is the lowest priority.

Incrementing Bursts

The interconnect converts all input INCR bursts that complete within a single output data width to an INCR1 burst of the minimum SIZE possible, and packs all INCR bursts into INCR bursts of the optimal size possible for maximum data throughout.

Wrapping Bursts

All WRAP bursts are either passed through unconverted as WRAP bursts, or converted to one or two INCR bursts of the output bus. The interconnect converts input WRAP bursts that have a total payload less than the output data width to a single INCR burst.

Fixed Bursts

All FIXED bursts pass through unconverted.

Bypass Merge

Bypass merge is accessible through the GPV registers and is only accessible to secure masters. If the programmable bit `bypass_merge` is enabled, the interconnect does not alter any transactions that could pass through legally without alteration.

Downsizing Data Width Function

The downsizing function reduces the data width of a transaction to match the optimal data width at the destination. Downsizing does not merge multiple-transaction data narrower than the destination bus if the transactions are marked as noncacheable.

The downsizing function reduces the data width by the following ratios:

- 2:1
- 4:1

Incrementing Bursts

The interconnect converts INCR bursts that fall within the maximum payload size of the output data bus to a single INCR burst. It converts INCR bursts that are greater than the maximum payload size of the output data bus to multiple INCR bursts.

INCR bursts with a size that matches the output data width pass through unconverted.

The interconnect packs INCR bursts with a SIZE smaller than the output data width to match the output width whenever possible, using the upsizing function.

Related Information

[Upsizing Data Width Function](#) on page 8-19

For more information about AXI terms such as DECERR, WRAP, and INCR, refer to the *AMBA AXI Protocol Specification v1.0*, which you can download from the Arm website.

Wrapping Bursts

The interconnect always converts WRAP bursts to WRAP bursts of twice the length, up to the output data width maximum size of WRAP16, and treats the WRAP burst as two INCR bursts that can each be converted into one or more INCR bursts.

Fixed Bursts

The interconnect converts FIXED bursts to one or more INCR1 or INCRn bursts depending on the downsize ratio.

Bypass Merge

Bypass merge is accessible through the GPV registers and is only accessible to secure masters. If the programmable bit `bypass_merge` in the `fn_mod2` register is enabled, the interconnect does not perform any packing of beats to match the optimal size for maximum throughput, up to the output data width size.

If an exclusive transaction is split into multiple transactions at the output of the downsizing function, the exclusive flag is removed and the master never receives an EXOKAY response. Response priority is the same as for the upsizing function.

Related Information

[Upsizing Data Width Function](#) on page 8-19

For more information about AXI terms such as DECERR, WRAP, and INCR, refer to the *AMBA AXI Protocol Specification v1.0*, which you can download from the Arm website.

Lock Support

Lock is not supported by the interconnect. For atomic accesses, masters can perform exclusive accesses when sharing data located in the HPS SDRAM.

Related Information

[SDRAM Controller Subsystem](#) on page 11-1

For more information about exclusive access support, refer to the *SDRAM Controller Subsystem* chapter.

FIFO Buffers and Clock Crossing

The interconnect provides FIFO buffers in the following locations:

- On interfaces to all HPS master and slaves except onchip RAM and boot ROM
- Between subswitches

In addition to buffering, these FIFOs also provide clock domain crossing where masters and slaves operate at a different clock frequency from the switch they connect to.

Data Release Mechanism

For system interconnect ports with data FIFO buffers whose depth is greater than zero, you can set a write tidemark function, `wr_tidemark`. This tidemark level stalls the release of the transaction until one of the following situations occurs:

- The system interconnect receives the `WLAST` beat of a burst.
- The write data FIFO buffer becomes full.
- The number of occupied slots in the write data FIFO buffer exceeds the write tidemark.

Related Information

- [System Interconnect Master Properties](#) on page 8-15
Indicates which master interfaces have data FIFO buffers with a nonzero depth
- [Interconnect Slave Properties](#) on page 8-17
Indicates which slave interfaces have data FIFO buffers with a nonzero depth

System Interconnect Resets

The system interconnect has one reset signal. The reset manager drives this signal to the system interconnect on a cold or warm reset.

Related Information

[Reset Manager](#) on page 4-1

System Interconnect Address Map and Register Definitions

This section lists the system interconnect register address map and describes the registers.

Note: System interconnect slaves are available for connection from peripheral masters. System interconnect masters connect to peripheral slaves. This terminology is the reverse of conventional terminology used in Platform Designer (Standard).

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
The base addresses of all modules are also listed in the *Introduction to the Hard Processor* chapter.
- [Arria V Address Map and Register Definitions](#)
Web-based address map and register definitions

2021.05.07

av_5v4



Subscribe



Send Feedback

This chapter describes the bridges in the hard processor system (HPS) used to communicate data between the FPGA fabric and HPS logic. The bridges use the Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) protocol, and are based on the AMBA Network Interconnect (NIC-301).

The HPS contains the following HPS-FPGA bridges:

- FPGA-to-HPS Bridge
- HPS-to-FPGA Bridge
- Lightweight HPS-to-FPGA Bridge

Related Information

- [Functional Description of the FPGA-to-HPS Bridge](#) on page 9-4
- [HPS-to-FPGA Bridge Clocks and Resets](#) on page 9-14
- [Lightweight HPS-to-FPGA Bridge Clocks and Resets](#) on page 9-14

Features of the HPS-FPGA Bridges

The HPS-FPGA bridges allow masters in the FPGA fabric to communicate with slaves in the HPS logic and vice versa. For example, if you implement memories or peripherals in the FPGA fabric, HPS components such as the MPU can access them. Components implemented in the FPGA fabric, such as the Nios[®] II processor, can also access memories and peripherals in the HPS logic.

Table 9-1: HPS-FPGA Bridge Features

Feature	FPGA-to-HPS Bridge	HPS-to-FPGA Bridge	Lightweight HPS-to-FPGA Bridge
Supports the AMBA AXI3 interface protocol	Y	Y	Y
Implements clock crossing and manages the transfer of data across the clock domains in the HPS logic and the FPGA fabric	Y	Y	Y

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

Feature	FPGA-to-HPS Bridge	HPS-to-FPGA Bridge	Lightweight HPS-to-FPGA Bridge
Performs data width conversion between the HPS logic and the FPGA fabric	Y	Y	Y
Allows configuration of FPGA interface widths at instantiation time	Y	Y	N

Each bridge consists of a master-slave pair with one interface exposed to the FPGA fabric and the other exposed to the HPS logic. The FPGA-to-HPS bridge exposes an AXI slave interface that you can connect to AXI master or Avalon-MM interfaces in the FPGA fabric. The HPS-to-FPGA and lightweight HPS-to-FPGA bridges expose an AXI master interface that you can connect to AXI or Avalon-MM slave interfaces in the FPGA fabric.

Related Information

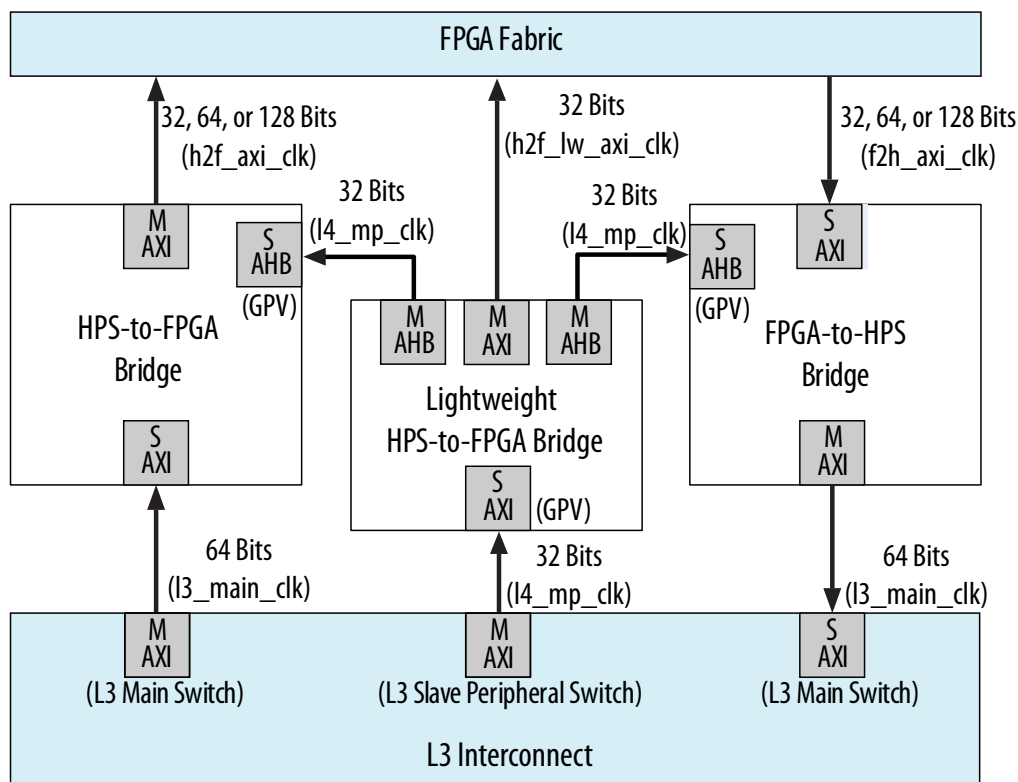
[AXI Bridges](#) on page 26-7

Information about configuring the AXI bridges

HPS-FPGA Bridges Block Diagram and System Integration

Figure 9-1: HPS-FPGA Bridge Connectivity

The following figure shows the HPS-FPGA bridges in the context of the FPGA fabric and the L3 interconnect to the HPS. Each master (M) and slave (S) interface is shown with its data width(s). The clock domain for each interconnect is shown in parentheses.



The HPS-to-FPGA bridge is mastered by the level 3 (L3) main switch and the lightweight HPS-to-FPGA bridge is mastered by the L3 slave peripheral switch.

The FPGA-to-HPS bridge masters the L3 main switch, allowing any master implemented in the FPGA fabric to access most slaves in the HPS. For example, the FPGA-to-HPS bridge can access the accelerator coherency port (ACP) of the Cortex-A9 MPU subsystem to perform cache-coherent accesses to the SDRAM subsystem.

All three bridges contain global programmers view (GPV) registers. The GPV registers control the behavior of the bridge. Access to the GPV registers of all three bridges is provided through the lightweight HPS-to-FPGA bridge.

Related Information

- [Clocks and Resets](#) on page 9-14
- [Functional Description of the System Interconnect](#)
Detailed information about connectivity, such as which masters have access to each bridge

Functional Description of the HPS-FPGA Bridges

The Global Programmers View

The HPS-to-FPGA bridge includes a set of registers called the GPV. The GPV provides settings to control the bridge properties and behavior. Access to the GPV registers of all three bridges is provided through the lightweight HPS-to-FPGA bridge.

The GPV registers can only be accessed by secure masters in the HPS or the FPGA fabric.

Functional Description of the FPGA-to-HPS Bridge

The FPGA-to-HPS bridge provides access to the peripherals and memory in the HPS. This access is available to any master implemented in the FPGA fabric. You can configure the bridge slave, which is exposed to the FPGA fabric, to support 32-, 64-, or 128-bit data. The master interface of the bridge, connected to the L3 interconnect, has a data width of 64 bits.

Table 9-2: FPGA-to-HPS Bridge Properties

The following table lists the properties of the FPGA-to-HPS bridge, including the configurable slave interface exposed to the FPGA fabric.

Bridge Property	FPGA Slave Interface	L3 Master Interface
Data width ⁽²⁰⁾	32, 64, or 128 bits	64 bits
Clock domain	f2h_axi_clk	l3_main_clk
Byte address width	32 bits	32 bits
ID width	8 bits	8 bits
Read acceptance	16 transactions	16 transactions
Write acceptance	16 transactions	16 transactions
Total acceptance	32 transactions	32 transactions

The FPGA-to-HPS bridge address map contains a GPV. The GPV registers provide settings that adjust the bridge slave properties when the FPGA slave interface is configured to be 32 or 128 bits wide. The slave issuing capability can be adjusted, through the `fn_mod` register, to allow one or multiple transactions to be outstanding in the HPS. The slave bypass merge feature can also be enabled, through the `bypass_merge` bit in the `fn_mod2` register. This feature ensures that the upsizing and downsizing logic does not alter any transactions when the FPGA slave interface is configured to be 32 or 128 bits wide.

Note: It is critical to provide the correct `l4_mp_clk` clock to support access to the GPV, as described in "GPV Clocks".

⁽²⁰⁾ The bridge slave data width is user-configurable at the time you instantiate the HPS component in your system.

Related Information

- [The Global Programmers View](#) on page 9-4
- [GPV Clocks](#) on page 9-15

FPGA-to-HPS Access to ACP

When the error correction code (ECC) option is enabled in the level 2 (L2) cache controller, all accesses from the FPGA-to-HPS bridge to the ACP must be 64 bits wide and aligned on 8-byte boundaries after up- or downsizing takes place. Ensure that the address alignment is a multiple of 8 bytes and that (burst size) × (burst length) is a multiple of 8 bytes.

FPGA-to-HPS Bridge Slave Signals

The FPGA-to-HPS bridge slave address channels support user-sideband signals, routed to the ACP in the MPU subsystem. All the signals have a fixed width except the data and write strobes for the read and write data channels. The variable width signals depend on the data width setting of the bridge.

The following tables list all the signals exposed by the FPGA-to-HPS slave interface to the FPGA fabric.

Table 9-3: FPGA-to-HPS Bridge Slave Write Address Channel Signals

Signal	Width	Direction	Description
AWID	8 bits	Input	Write address ID
AWADDR	32 bits	Input	Write address
AWLEN	4 bits	Input	Burst length
AWSIZE	3 bits	Input	Burst size
AWBURST	2 bits	Input	Burst type
AWLOCK	2 bits	Input	Lock type—Valid values are 00 (normal access) and 01 (exclusive access)
AWCACHE	4 bits	Input	Cache policy type
AWPROT	3 bits	Input	Protection type
AWVALID	1 bit	Input	Write address channel valid
AWREADY	1 bit	Output	Write address channel ready
AWUSER	5 bits	Input	User sideband signals

Table 9-4: FPGA-to-HPS Bridge Slave Write Data Channel Signals

Signal	Width	Direction	Description
WID	8 bits	Input	Write ID

Signal	Width	Direction	Description
WDATA	32, 64, or 128 bits	Input	Write data
WSTRB	4, 8, or 16 bits	Input	Write data strobes
WLAST	1 bit	Input	Write last data identifier
WVALID	1 bit	Input	Write data channel valid
WREADY	1 bit	Output	Write data channel ready

Table 9-5: FPGA-to-HPS Bridge Slave Write Response Channel Signals

Signal	Width	Direction	Description
BID	8 bits	Output	Write response ID
BRESP	2 bits	Output	Write response
BVALID	1 bit	Output	Write response channel valid
BREADY	1 bit	Input	Write response channel ready

Table 9-6: FPGA-to-HPS Bridge Slave Read Address Channel Signals

Signal	Width	Direction	Description
ARID	8 bits	Input	Read address ID
ARADDR	32 bits	Input	Read address
ARLEN	4 bits	Input	Burst length
ARSIZE	3 bits	Input	Burst size
ARBURST	2 bits	Input	Burst type
ARLOCK	2 bits	Input	Lock type—Valid values are 00 (normal access) and 01 (exclusive access)
ARCACHE	4 bits	Input	Cache policy type
ARPROT	3 bits	Input	Protection type
ARVALID	1 bit	Input	Read address channel valid
ARREADY	1 bit	Output	Read address channel ready

Signal	Width	Direction	Description
ARUSER	5 bits	Input	Read user sideband signals

Table 9-7: FPGA-to-HPS Bridge Slave Read Data Channel Signals

Signal	Width	Direction	Description
RID	8 bits	Output	Read ID
RDATA	32, 64, or 128 bits	Output	Read data
RRESP	2 bits	Output	Read response
RLAST	1 bit	Output	Read last data identifier
RVALID	1 bit	Output	Read data channel valid
RREADY	1 bit	Input	Read data channel ready

Functional Description of the HPS-to-FPGA Bridge

The HPS-to-FPGA bridge provides a configurable-width, high-performance master interface to the FPGA fabric. The bridge provides most masters in the HPS with access to logic, peripherals, and memory implemented in the FPGA. The effective size of the address space is 0x3FFF0000, or 1 gigabyte (GB) minus the 64 megabytes (MB) occupied by peripherals, lightweight HPS-to-FPGA bridge, on-chip RAM, and boot ROM in the HPS. You can configure the bridge master exposed to the FPGA fabric for 32-, 64-, or 128-bit data. The amount of address space exposed to the MPU subsystem can also be reduced through the L2 cache address filtering mechanism.

The slave interface of the bridge in the HPS logic has a data width of 64 bits. The bridge provides width adaptation and clock crossing logic that allows the logic in the FPGA to operate in any clock domain, asynchronous from the HPS.

Note: The HPS-to-FPGA bridge is accessed if the MPU boots from the FPGA. Before the MPU boots from the FPGA, the FPGA portion of the SoC device must be configured, and the HPS-to-FPGA bridge must be remapped into addressable space.

Table 9-8: HPS-to-FPGA Bridge Properties

The following table lists the properties of the HPS-to-FPGA bridge, including the configurable master interface exposed to the FPGA fabric.

Bridge Property	L3 Slave Interface	FPGA Master Interface
Data width ⁽²¹⁾	64 bits	32, 64, or 128 bits
Clock domain	l3_main_clk	h2f_axi_clk

⁽²¹⁾ The bridge master data width is user-configurable at the time you instantiate the HPS component in your system.

Bridge Property	L3 Slave Interface	FPGA Master Interface
Byte address width	32 bits	30 bits
ID width	12 bits	12 bits
Read acceptance	16 transactions	16 transactions
Write acceptance	16 transactions	16 transactions
Total acceptance	32 transactions	32 transactions

The HPS-to-FPGA bridge's GPV, described in "The Global Programmers View", provides settings to adjust the bridge master properties. The master issuing capability can be adjusted, through the `fn_mod` register, to allow one or multiple transactions to be outstanding in the FPGA fabric. The master bypass merge feature can also be enabled, through the `bypass_merge` bit in the `fn_mod2` register. This feature ensures that the upsizing and downsizing logic does not alter any transactions when the FPGA master interface is configured to be 32 or 128 bits wide.

Note: It is critical to provide the correct `l4_mp_clk` clock to support access to the GPV, as described in "GPV Clocks".

Related Information

- [The Global Programmers View](#) on page 9-4
- [GPV Clocks](#) on page 9-15
- [Functional Description of the System Interconnect](#)
Detailed information about connectivity, such as which masters have access to each bridge
- [Cortex-A9 Microprocessor Unit Subsystem](#)
Details about L2 cache address filtering
- [AXI Bridges](#) on page 26-7
Information about configuring the AXI bridges

HPS-to-FPGA Bridge Master Signals

All the HPS-to-FPGA bridge master signals have a fixed width except the data and write strobes for the read and write data channels. The variable-width signals depend on the data width setting of the bridge interface exposed to the FPGA logic.

The following tables list all the signals exposed by the HPS-to-FPGA master interface to the FPGA fabric.

Table 9-9: HPS-to-FPGA Bridge Master Write Address Channel Signals

Signal	Width	Direction	Description
AWID	12 bits	Output	Write address ID
AWADDR	30 bits	Output	Write address
AWLEN	4 bits	Output	Burst length

Signal	Width	Direction	Description
AWSIZE	3 bits	Output	Burst size
AWBURST	2 bits	Output	Burst type
AWLOCK	2 bits	Output	Lock type—Valid values are 00 (normal access) and 01 (exclusive access)
AWCACHE	4 bits	Output	Cache policy type
AWPROT	3 bits	Output	Protection type
AWVALID	1 bit	Output	Write address channel valid
AWREADY	1 bit	Input	Write address channel ready

Table 9-10: HPS-to-FPGA Bridge Master Write Data Channel Signals

Signal	Width	Direction	Description
WID	12 bits	Output	Write ID
WDATA	32, 64, or 128 bits	Output	Write data
WSTRB	4, 8, or 16 bits	Output	Write data strobes
WLAST	1 bit	Output	Write last data identifier
WVALID	1 bit	Output	Write data channel valid
WREADY	1 bit	Input	Write data channel ready

Table 9-11: HPS-to-FPGA Bridge Master Write Response Channel Signals

Signal	Width	Direction	Description
BID	12 bits	Input	Write response ID
BRESP	2 bits	Input	Write response
BVALID	1 bit	Input	Write response channel valid
BREADY	1 bit	Output	Write response channel ready

Table 9-12: HPS-to-FPGA Bridge Master Read Address Channel Signals

Signal	Width	Direction	Description
ARID	12 bits	Output	Read address ID
ARADDR	30 bits	Output	Read address
ARLEN	4 bits	Output	Burst length
ARSIZE	3 bits	Output	Burst size
ARBURST	2 bits	Output	Burst type
ARLOCK	2 bits	Output	Lock type—Valid values are 00 (normal access) and 01 (exclusive access)
ARCACHE	4 bits	Output	Cache policy type
ARPROT	3 bits	Output	Protection type
ARVALID	1 bit	Output	Read address channel valid
ARREADY	1 bit	Input	Read address channel ready

Table 9-13: HPS-to-FPGA Bridge Master Read Data Channel Signals

Signal	Width	Direction	Description
RID	12 bits	Input	Read ID
RDATA	32, 64, or 128 bits	Input	Read data
RRESP	2 bits	Input	Read response
RLAST	1 bit	Input	Read last data identifier
RVALID	1 bit	Input	Read data channel valid
RREADY	1 bit	Output	Read data channel ready

Functional Description of the Lightweight HPS-to-FPGA Bridge

The lightweight HPS-to-FPGA bridge provides a lower-performance interface to the FPGA fabric. This interface is useful for accessing the control and status registers of soft peripherals. The bridge provides a 2 MB address space and access to logic, peripherals, and memory implemented in the FPGA fabric. The MPU subsystem, direct memory access (DMA) controller, and debug access port (DAP) can use the lightweight HPS-to-FPGA bridge to access the FPGA fabric or GPV. Master interfaces in the FPGA fabric can also use the lightweight HPS-to-FPGA bridge to access the GPV registers in all three bridges.

The bridge master exposed to the FPGA fabric has a fixed data width of 32 bits. The slave interface of the bridge in the HPS logic has a fixed data width of 32 bits.

Use the lightweight HPS-to-FPGA bridge as a secondary, lower-performance master interface to the FPGA fabric. With a fixed width and a smaller address space, the lightweight bridge is useful for low-bandwidth traffic, such as memory-mapped register accesses to FPGA peripherals. This approach diverts traffic from the high-performance HPS-to-FPGA bridge, and can improve both register access latency and overall system performance.

Table 9-14: Lightweight HPS-to-FPGA Bridge Properties

This table lists the properties of the lightweight HPS-to-FPGA bridge, including the master interface exposed to the FPGA fabric.

Bridge Property	L3 Slave Interface	FPGA Master Interface
Data width	32 bits	32 bits
Clock domain	l4_mp_clk	h2f_lw_axi_clk
Byte address width	32 bits	21 bits
ID width	12 bits	12 bits
Read acceptance	16 transactions	16 transactions
Write acceptance	16 transactions	16 transactions
Total acceptance	32 transactions	32 transactions

The lightweight HPS-to-FPGA bridge has three master interfaces. The master interface connected to the FPGA fabric provides a lightweight interface from the HPS to custom logic in the FPGA fabric. The two other master interfaces, connected to the HPS-to-FPGA and FPGA-to-HPS bridges, allow you to access the GPV registers for each bridge.

The lightweight HPS-to-FPGA bridge also has a set of registers GPV to control the behavior of its four interfaces (one slave and three masters).

The GPV allows you to set the bridge's issuing capabilities to support single or multiple transactions. The GPV also lets you set a write tidemark through the `wr_tidemark` register, to control how much data is buffered in the bridge before data is written to slaves in the FPGA fabric.

Note: It is critical to provide correct clock settings for the lightweight HPS-to-FPGA bridge, even if your design does not use this bridge. The `l4_mp_clk` clock is required for GPV access on the HPS-to-FPGA and FPGA-to-HPS bridges.

Related Information

- [HPS-FPGA Bridges Block Diagram and System Integration](#) on page 9-3
Figure showing the lightweight HPS-to-FPGA bridge's three master interfaces
- [The Global Programmers View](#) on page 9-4
Includes a description of the lightweight HPS-to-FPGA bridge GPV

- **Functional Description of the System Interconnect**
Detailed information about connectivity, such as which masters have access to each bridge
- **AXI Bridges** on page 26-7
Information about configuring the AXI bridges

Lightweight HPS-to-FPGA Bridge Master Signals

All the lightweight HPS-to-FPGA bridge master signals have a fixed width.

The following tables list all the signals exposed by the lightweight HPS-to-FPGA master interface to the FPGA fabric.

Table 9-15: Lightweight HPS-to-FPGA Bridge Master Write Address Channel Signals

Signal	Width	Direction	Description
AWID	12 bits	Output	Write address ID
AWADDR	21 bits	Output	Write address
AWLEN	4 bits	Output	Burst length
AWSIZE	3 bits	Output	Burst size
AWBURST	2 bits	Output	Burst type
AWLOCK	2 bits	Output	Lock type—Valid values are 00 (normal access) and 01 (exclusive access)
AWCACHE	4 bits	Output	Cache policy type
AWPROT	3 bits	Output	Protection type
AWVALID	1 bit	Output	Write address channel valid
AWREADY	1 bit	Input	Write address channel ready

Table 9-16: Lightweight HPS-to-FPGA Bridge Master Write Data Channel Signals

Signal	Width	Direction	Description
WID	12 bits	Output	Write ID
WDATA	32 bits	Output	Write data
WSTRB	4 bits	Output	Write data strobes
WLAST	1 bit	Output	Write last data identifier
WVALID	1 bit	Output	Write data channel valid

Signal	Width	Direction	Description
WREADY	1 bit	Input	Write data channel ready

Table 9-17: Lightweight HPS-to-FPGA Bridge Master Write Response Channel Signals

Signal	Width	Direction	Description
BID	12 bits	Input	Write response ID
BRESP	2 bits	Input	Write response
BVALID	1 bit	Input	Write response channel valid
BREADY	1 bit	Output	Write response channel ready

Table 9-18: Lightweight HPS-to-FPGA Bridge Master Read Address Channel Signals

Signal	Width	Direction	Description
ARID	12 bits	Output	Read address ID
ARADDR	21 bits	Output	Read address
ARLEN	4 bits	Output	Burst length
ARSIZE	3 bits	Output	Burst size
ARBURST	2 bits	Output	Burst type
ARLOCK	2 bits	Output	Lock type—Valid values are 00 (normal access) and 01 (exclusive access)
ARCACHE	4 bits	Output	Cache policy type
ARPROT	3 bits	Output	Protection type
ARVALID	1 bit	Output	Read address channel valid
ARREADY	1 bit	Input	Read address channel ready

Table 9-19: Lightweight HPS-to-FPGA Bridge Master Read Data Channel Signals

Signal	Width	Direction	Description
RID	12 bits	Input	Read ID
RDATA	32 bits	Input	Read data

Signal	Width	Direction	Description
RRESP	2 bits	Input	Read response
RLAST	1 bit	Input	Read last data identifier
RVALID	1 bit	Input	Read data channel valid
RREADY	1 bit	Output	Read data channel ready

Clocks and Resets

FPGA-to-HPS Bridge Clocks and Resets

The master interface of the bridge in the HPS logic operates in the `l3_main_clk` clock domain. The slave interface exposed to the FPGA fabric operates in the `f2h_axi_clk` clock domain provided by the user logic. The bridge provides clock crossing logic that allows the logic in the FPGA to operate in any clock domain, asynchronous from the HPS.

The FPGA-to-HPS bridge has one reset signal, `fpga2hps_bridge_rst_n`. The reset manager drives this signal to the FPGA-to-HPS bridge on a cold or warm reset.

Related Information

- [Clock Manager](#) on page 3-1
- [HPS Component Interfaces](#) on page 27-1
Information about the HPS-FPGA bridge clock interfaces

HPS-to-FPGA Bridge Clocks and Resets

The master interface into the FPGA fabric operates in the `h2f_axi_clk` clock domain. The `h2f_axi_clk` clock is provided by user logic. The slave interface of the bridge in the HPS logic operates in the `l3_main_clk` clock domain. The bridge provides clock crossing logic that allows the logic in the FPGA to operate in any clock domain, asynchronous from the HPS.

The HPS-to-FPGA bridge has one reset signal, `hps2fpga_bridge_rst_n`. The reset manager drives this signal to the HPS-to-FPGA bridge on a cold or warm reset.

Related Information

- [Clock Manager](#) on page 3-1
- [HPS Component Interfaces](#) on page 27-1
Information about the HPS-FPGA bridge clock interfaces

Lightweight HPS-to-FPGA Bridge Clocks and Resets

The master interface into the FPGA fabric operates in the `h2f_lw_axi_clk` clock domain. The clock is provided by custom logic in the FPGA fabric. The slave interface of the bridge in the HPS logic operates in the `l4_mp_clk` clock domain. The bridge provides clock crossing logic that allows the logic in the FPGA to operate in any clock domain, asynchronous from the HPS.

The lightweight HPS-to-FPGA bridge has one reset signal, `lw_hps2fpga_bridge_rst_n`. The reset manager drives this signal to the lightweight HPS-to-FPGA bridge on a cold or warm reset.

Related Information

- [Clock Manager](#) on page 3-1
- [HPS Component Interfaces](#) on page 27-1
Information about the HPS-FPGA bridge clock interfaces

Taking HPS-FPGA Bridges Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

Related Information

[Modules Requiring Software Deassert](#) on page 4-9
Reset register names

GPV Clocks

The FPGA-to-HPS and HPS-to-FPGA bridges have GPV slave interfaces, mastered by the lightweight HPS-to-FPGA bridge. These interfaces operate in the `l4_mp_clk` clock domain. Even if you do not use the lightweight HPS-to-FPGA bridge in your FPGA design, you must ensure that a valid `l4_mp_clk` clock is being generated, so that the GPV registers in the HPS-to-FPGA and FPGA-to-HPS bridges can be programmed. The GPV logic in all three bridges is in the `l4_mp_clk` domain.

Related Information

[The Global Programmers View](#) on page 9-4

Data Width Sizing

The HPS-to-FPGA and FPGA-to-HPS bridges allow 32-, 64-, and 128-bit interfaces to be exposed to the FPGA fabric. For 32-bit and 128-bit interfaces, the bridge performs data width conversion to the fixed 64-bit interface within the HPS. This conversion is called *upsizing* in the case of data being converted from a 64-bit interface to a 128-bit interface. It is called *downsizing* in the case of data being converted from a 64-bit interface to a 32-bit interface. If an exclusive access is split into multiple transactions, the transactions lose their exclusive access information.

During the upsizing or downsizing process, transactions can also be resized using a data merging technique. For example, in the case of a 32-bit to 64-bit upsizing, if the size of each beat entering the bridge's 32-bit interface is only two bytes, the bridge can merge up to four beats to form a single 64-bit beat. Similarly, in the case of a 128-bit to 64-bit downsizing, if the size of each beat entering the bridge's 128-bit interface is only four bytes, the bridge can merge two beats to form a single 64-bit beat.

The bridges do not perform transaction merging for accesses marked as noncacheable.

Note: You can set the `bypass_merge` bit in the GPV to prevent the bridge from merging data and responses. If the bridge merges multiple responses into a single response, that response is the one with the highest priority. The response types have the following priorities:

1. DECERR
2. SLVERR
3. OKAY

HPS-FPGA Bridges Address Map and Register Definitions

The address map and register definitions for the HPS-FPGA bridges consist of the following regions:

- FPGA-to-HPS Bridge Module
- HPS-to-FPGA Bridge Module
- Lightweight HPS-to-FPGA Bridge Module
- FPGA Slaves Accessed via Lightweight HPS-to-FPGA AXI Bridge

Related Information

- [HPS Peripheral Region Address Map](#) on page 2-17
Lists the base addresses of all modules
- [Arria V Address Map and Register Definitions](#)
Web-based address map and register definitions



2021.05.07

av_5v4



Subscribe



Send Feedback

CoreSight systems provide all the infrastructure you require to debug, monitor, and optimize the performance of a complete HPS design. CoreSight technology addresses the requirement for a multicore debug and trace solution with high bandwidth for whole systems beyond the processor core.

CoreSight technology provides the following features:

- Cross-trigger support between SoC subsystems
- High data compression
- Multi-source trace in a single stream
- Standard programming models for standard tool support

The hard processor system (HPS) debug infrastructure provides visibility and control of the HPS modules, the Arm Cortex-A9 microprocessor unit (MPU) subsystem, and user logic implemented in the FPGA fabric. The debug system design incorporates Arm CoreSight components.

Details of the Arm CoreSight debug components can be viewed by clicking on the following related information links:

Related Information

- [Debug Access Port](#) on page 10-4
- [System Trace Macrocell](#) on page 10-4
- [Trace Funnel](#) on page 10-5
- [Embedded Trace FIFO](#) on page 10-5
- [AMBA Trace Bus Replicator](#) on page 10-6
- [Embedded Trace Router](#) on page 10-6
- [Trace Port Interface Unit](#) on page 10-6
- [Embedded Cross Trigger System](#) on page 10-6
- [Program Trace Macrocell](#) on page 10-11

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

Features of CoreSight Debug and Trace

The CoreSight debug and trace system offers the following features:

- Real-time program flow instruction trace through a separate Program Trace Macrocell (PTM) for each processor
- Host debugger JTAG interface
- Connections for cross-trigger and STM-to-FPGA interfaces, which enable soft IP generation of triggers and system trace messages
- External trace interface through Trace Port Interface Unit (TPIU) for trace analysis tools
- Custom message injection through the System Trace Macrocell (STM) into the trace stream for delivery to a host debugger
- STM and PTM trace sources multiplexed into a single stream through the Trace Funnel
- Capability to route trace data to any slave accessible to the Embedded Trace Router (ETR) AXI master connected to the level 3 (L3) interconnect
- Capability for the following components to trigger each other through the embedded cross-trigger system:
 - Cortex-A9 PTM-0
 - Cortex-A9-A9 PTM-1
 - STM
 - Embedded Trace FIFO (ETF)
 - ETR
 - TPIU
 - csCross Trigger Interface (CTI)
 - FPGA-CTI
 - csCross Trigger Matrix (CTM)

Related Information

[Cross Trigger Interface](#) on page 10-8

Arm CoreSight Documentation

The following Arm CoreSight specifications and documentation provide a more thorough description of the Arm CoreSight components in the HPS debug system:

- *CoreSight Technology System Design Guide* (Arm DGI 0012D)
- *CoreSight Architecture Specification*
- *Embedded Cross Trigger Technical Reference Manual* (Arm DDI 0291A)
- *CoreSight Components Technical Reference Manual* (Arm DDI 0314H)
- *CoreSight System Trace Macrocell Technical Reference Manual* (Arm DDI 0444A)
- *System Trace Macrocell Programmers' Model Architecture Specification* (Arm IHI 0054)
- *CoreSight Trace Memory Controller Technical Reference Manual* (Arm DDI 0461B)

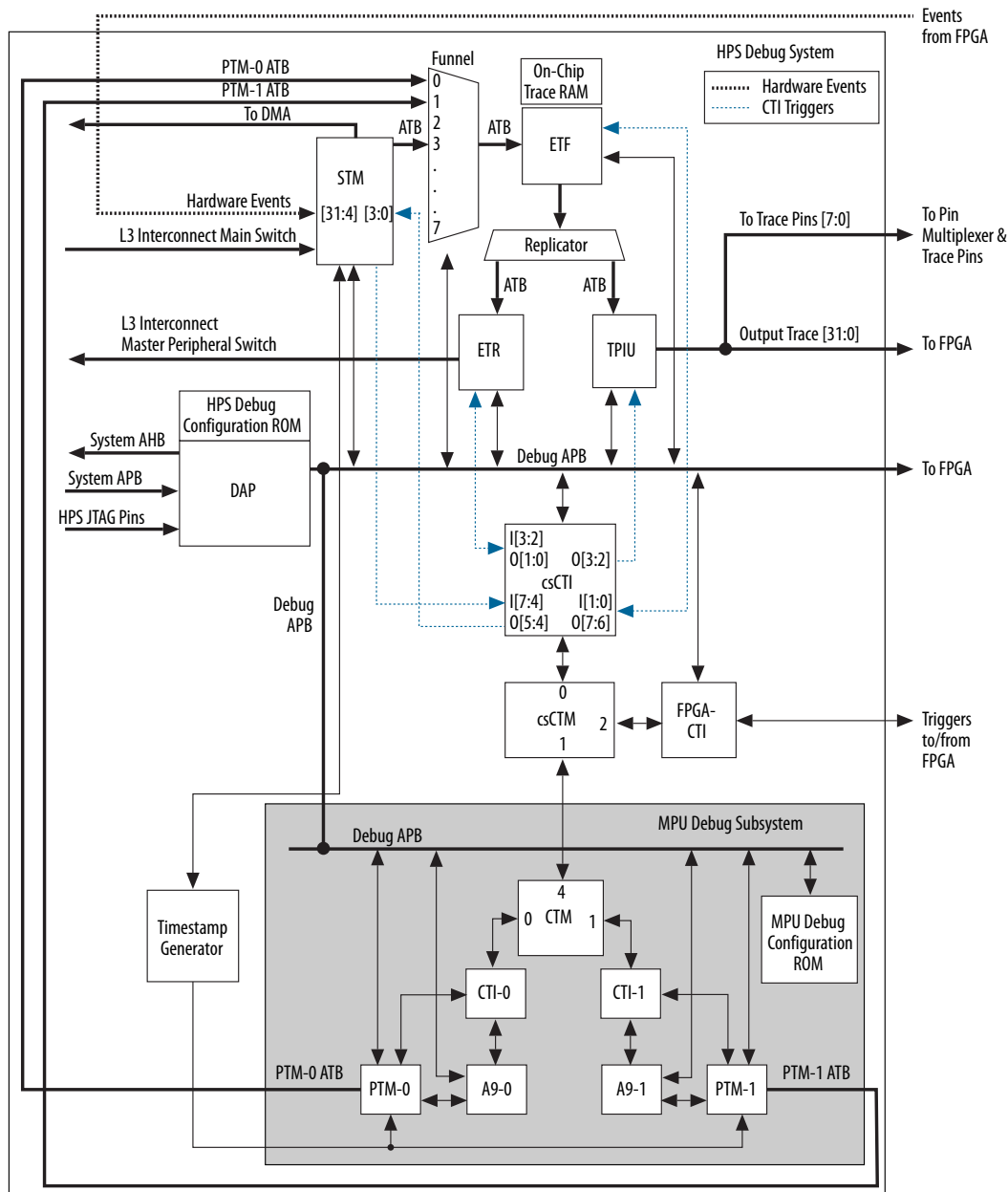
Related Information

Arm Infocenter

You can download these documents from the Arm Infocenter website.

CoreSight Debug and Trace Block Diagram and System Integration

Figure 10-1: HPS CoreSight Debug and Trace System Block Diagram



Functional Description of CoreSight Debug and Trace

Debug Access Port

The Debug Access Port (DAP) provides the necessary ports for a host debugger to connect to and communicate with the HPS through a JTAG interface connected to dedicated HPS pins that is independent of the JTAG for the FPGA. The JTAG interface provided with the DAP allows a host debugger to access various modules inside the HPS. Additionally, a debug monitor executing on either processor can access different HPS components by interfacing with the system Advanced Microcontroller Bus Architecture (AMBA) Advanced Peripheral Bus (APB) slave port of the DAP.

The system APB slave port occupies 2 MB of address space in the HPS. Both the JTAG port and system APB port have access to the debug APB master port of the DAP.

A host debugger can access any HPS memory-mapped resource in the system through the DAP system master port. Requests made over the DAP system master port are impacted by reads and writes to peripheral registers.

Note: The HPS JTAG interface does not support boundary scan tests (BST). To perform boundary scan testing on HPS I/Os, use the FPGA JTAG pins.

Related Information

- [CoreSight Debug and Trace Block Diagram and System Integration](#) on page 10-3
Shows CoreSight components connected to the debug APB
- [Arria V Device Handbook Volume 1: Device Interfaces and Integration](#)
For more information about boundary scan tests, refer to the "JTAG Boundary-Scan Testing in Arria V Devices" chapter.
- [Arm Infocenter](#)
Refer to the *CoreSight Components Technical Reference Manual* on the Arm Infocenter website.

System Trace Macrocell

The STM allows messages to be injected into the trace stream for delivery to the host debugger receiving the trace data. These messages can be sent through stimulus ports or the hardware EVENT interface. The STM allows these messages to be time stamped.

The STM provides an AMBA Advanced eXtensible Interface (AXI) slave interface used to create trace events. The interface can be accessed by the MPU subsystem, direct memory access (DMA) controller, and masters implemented as soft logic in the FPGA fabric through the FPGA-to-HPS bridge. The AXI slave interface supports three address segments, where each address segment is 16 MB and each segment supports up to 65536 channels. Each channel occupies 256 bytes of address space.

The STM also provides 32 hardware EVENT pins. The higher-order 28 pins (31:4) are connected to the FPGA fabric, allowing logic inside FPGA to insert messages into the trace stream. When the STM detects a rising edge on an EVENT pin, a message identifying the EVENT is inserted into the stream. The lower four EVENT pins (3:0) are connected to csCTI.

Related Information

- [HPS-FPGA Bridges](#) on page 9-1

- [Arm Infocenter](#)
For more information, refer to the *CoreSight System Trace Macrocell Technical Reference Manual* on the Arm Infocenter website.
- [csCTI](#) on page 10-19

Trace Funnel

The Trace Funnel is used to combine multiple trace streams into one trace stream. There are three trace streams that use the following funnel ports:

Table 10-1: Trace Stream Connections

Funnel Port	Description
0	The trace stream coming from PTM connected to CPU0 uses this port.
1	The trace stream coming from PTM connected to CPU1 uses this port.
2	Not connected.
3	The trace stream coming from STM uses this port.
4 .. 7	Not connected.

Related Information

- [Arm Infocenter](#)
Refer to the *CoreSight Components Technical Reference Manual* on the Arm Infocenter website.
- [Program Trace Macrocell](#) on page 10-11

CoreSight Trace Memory Controller

The CoreSight Trace Memory Controller (TMC) has three possible configurations:

- Embedded Trace FIFO (ETF)
- Embedded Trace Router (ETR)
- Embedded Trace Buffer (ETB)

ETB is not used in this device.

Related Information

[Arm Infocenter](#)

For more information, refer to the *CoreSight System Trace Memory Controller Technical Reference Manual* on the Arm Infocenter website.

Embedded Trace FIFO

The Trace Funnel output is sent to the ETF. The ETF is used as an elastic buffer between trace generators (STM, PTM) and trace destinations. The ETF stores up to 32 KB of trace data in the on-chip trace RAM.

Related Information**[Arm Infocenter](#)**

For more information, refer to the *CoreSight System Trace Memory Controller Technical Reference Manual* on the Arm Infocenter website.

Embedded Trace Router

The ETR can route trace data to the HPS on-chip RAM, the HPS SDRAM, and any memory in the FPGA fabric connected to the HPS-to-FPGA bridge. The ETR receives trace data from the AMBA Trace Bus Replicator. By default, the buffer to receive the trace data resides in SDRAM at offset 0x00100000 and is 32 KB. You can override this default configuration by programming registers in the ETR.

Related Information

- **[HPS-FPGA Bridges](#)** on page 9-1
- **[CoreSight Debug and Trace Programming Model](#)** on page 10-16
- **[Arm Infocenter](#)**

For more information, refer to the *CoreSight System Trace Memory Controller Technical Reference Manual* on the Arm Infocenter website.

AMBA Trace Bus Replicator

The AMBA Trace Bus Replicator broadcasts trace data from the ETF to the ETR and TPIU.

Related Information**[Arm Infocenter](#)**

Refer to the *CoreSight Components Technical Reference Manual* on the Arm Infocenter website.

Trace Port Interface Unit

The TPIU is a bridge between on-chip trace sources and an off-chip trace port. The TPIU receives trace data from the Trace Bus Replicator (Replicator) and drives the trace data to a trace port analyzer.

The trace output from the TPIU is software programmable and can be set to either 8 or 32 bits wide. The trace output is routed to an 8-bit HPS I/O interface and a 32-bit interface to the FPGA fabric. The trace data sent to the FPGA fabric can be transported off-chip using available serializer/deserializer (SERDES) resources in the FPGA.

Related Information**[Arm Infocenter](#)**

Refer to the *CoreSight Components Technical Reference Manual* on the Arm Infocenter website.

Embedded Cross Trigger System

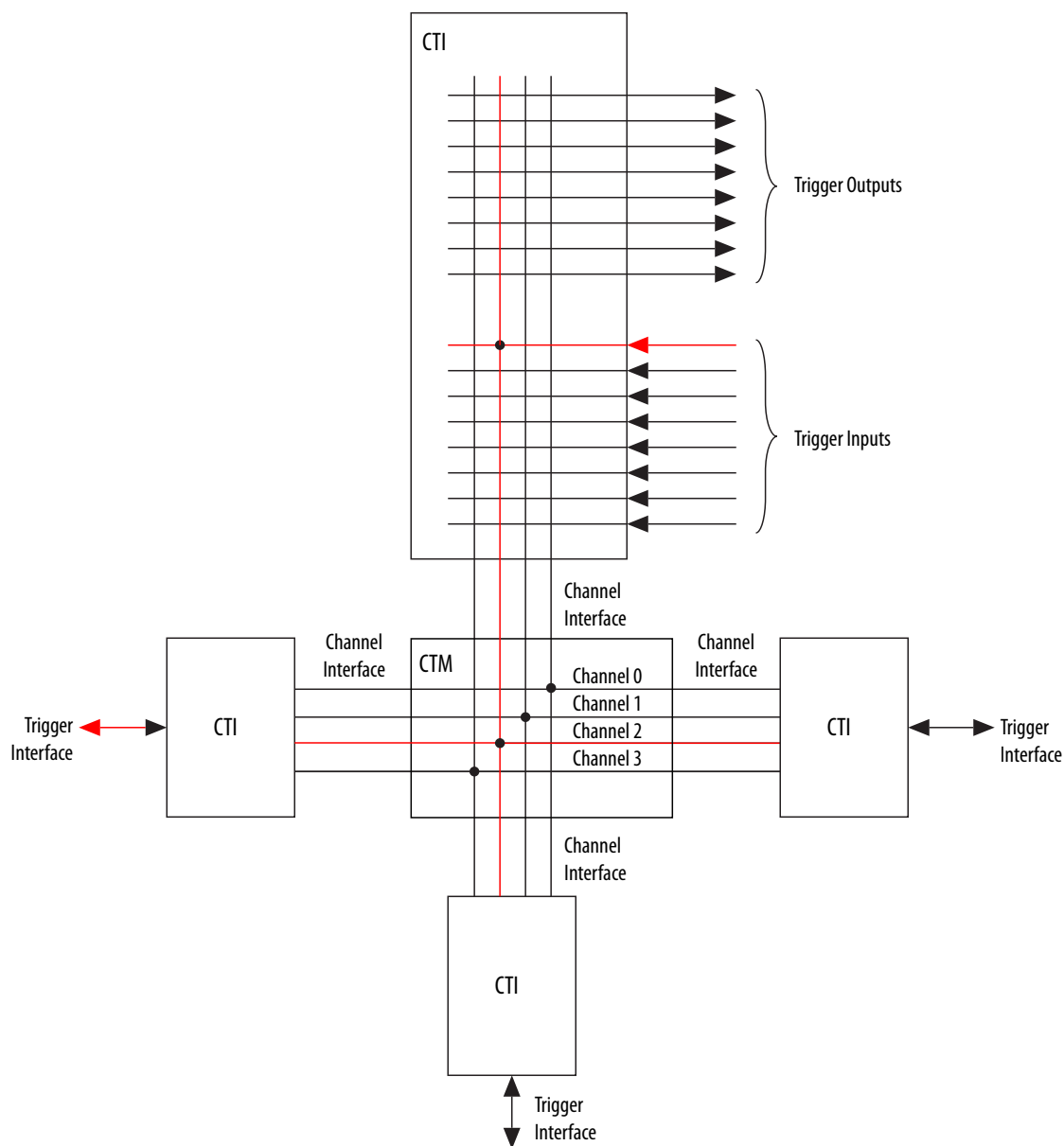
The ECT system provides a mechanism for the components listed in "Features of the CoreSight Debug and Trace" to trigger each other. The ECT consists of the following modules:

- Cross Trigger Interface (CTI)
- Cross Trigger Matrix (CTM)

Figure 10-2: Generic ECT System

The following figure shows an example of how CTIs and CTMs are used in a generic ECT setup. Though the signal travels through channel 2, it only enters and exits through trigger inputs and outputs you configure.

Note: The red line depicts an trigger input to one CTI generating a trigger output in another CTI.



Related Information

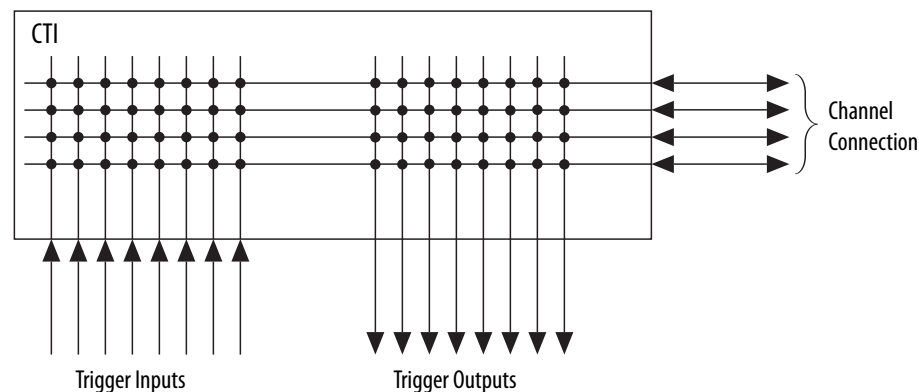
- [Cross Trigger Interface](#) on page 10-8
- [Features of CoreSight Debug and Trace](#) on page 10-2
- [Cross Trigger Matrix](#) on page 10-8

Cross Trigger Interface

CTIs allow trigger sources and sinks to interface with the ECT. Each CTI supports up to eight trigger inputs and eight trigger outputs, and is connected to a CTM. **Figure 10-3** shows the relationship of trigger inputs, trigger outputs, and CTM channels of a CTI.

Figure 10-3: CTI Connections

The following figure shows the eight potential trigger input and trigger output connections that are supported.



The HPS debug system contains the following CTIs:

- csCTI—performs cross triggering between the STM, ETF, ETR, and TPIU.
- FPGA-CTI—exposes the cross-triggering system to the FPGA fabric.
- CTI-0 and CTI-1—reside in the MPU debug subsystem. Each CTI is associated with a processor and the processor's PTM.

Cross Trigger Matrix

A CTM is a transport mechanism for triggers traveling from one CTI to one or more CTIs or CTMs. The HPS contains two CTMs. One CTM connects csCTI and FPGA-CTI; the other connects CTI-0 and CTI-1. The two CTMs are connected together, allowing triggers to be transmitted between the MPU debug subsystem, the debug system, and the FPGA fabric.

Each CTM has four ports and each port has four channels. Each CTM port can be connected to a CTI or another CTM.

Figure 10-4: CTM Channel Structure

The following figure shows the structure of a CTM channel. Paths inside the CTM are purely combinational.

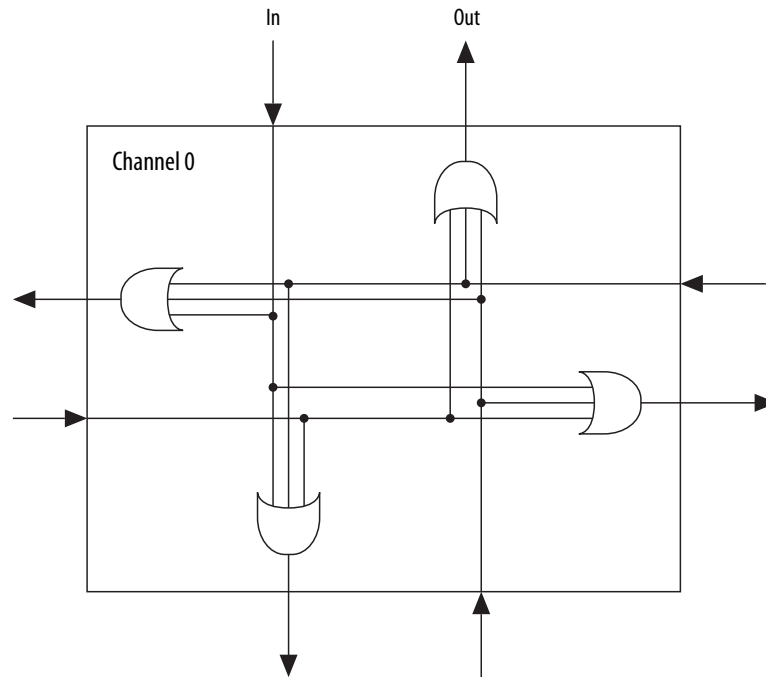
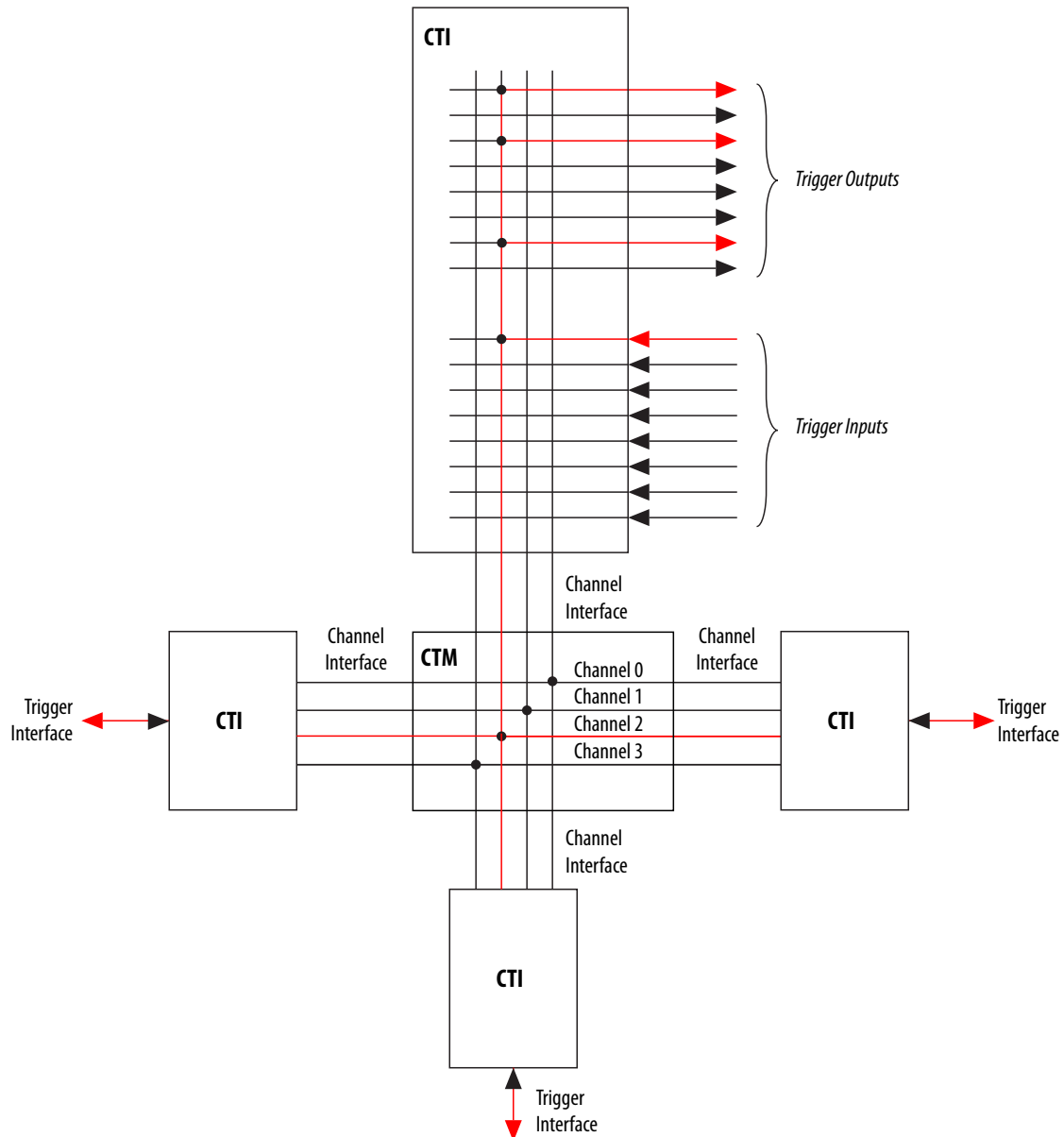


Figure 10-5: CTI Trigger Connections

Each CTI trigger input can be connected through a CTM to one or more trigger outputs under control by a debugger. The following figure shows an example of CTI trigger connections.

Note: The red lines depict the impact one trigger input can have on the entire system.



Related Information

[Arm Infocenter](#)

Refer to the *CoreSight Components Technical Reference Manual* on the Arm Infocenter website.

Program Trace Macrocell

The PTM performs real-time program flow instruction tracing and provides a variety of filters and triggers that can be used to trace specific portions of code.

The HPS contains two PTMs. Each PTM is paired with a processor and CTI. Trace data generated from the PTM can be transmitted off-chip using HPS pins, or to the FPGA fabric, where it can be pre-processed and transmitted off-chip using high-speed FPGA pins.

Related Information

[Arm Infocenter](#)

For more information, refer to the *CoreSight PTM-A9 Technical Reference Manual*.

HPS Debug APB Interface

The HPS can extend the CoreSight debug control bus into the FPGA fabric. The debug interface is an APB-compatible interface with built-in clock crossing.

Related Information

- [FPGA Interface](#) on page 10-11
- [HPS Component Interfaces](#) on page 27-1

FPGA Interface

The following components connect to the FPGA fabric. This section lists the signals from the debug system to the FPGA.

DAP

The DAP uses the system APB port to connect to the FPGA.

Table 10-2: DAP

The following table shows the signal description between DAP and FPGA.

Signal	Description
h2f_dbg_apb_PADDR	Address bus to system APB port, when PADDR
h2f_dbg_apb_PADDR31	Address bus to system APB port, when PADDR31
h2f_dbg_apb_PENABLE	Enable signal from system APB port
h2f_dbg_apb_PRDATA[32]	32-bit system APB port read data bus
h2f_dbg_apb_PREADY	Ready signal to system APB port
h2f_dbg_apb_PSEL	Select signal from system APB port
h2f_dbg_apb_PSLVERR	Error signal to system APB port
h2f_dbg_apb_PWDATA[32]	32-bit system APB port write data bus

Signal	Description
h2f_dbg_apb_PWRITE	Select whether read or write to system APB port 0 - System APB port read from DAP 1 - System APB Port write to DAP

STM

The STM has 28 event pins, `f2h_stm_hw_events[28]`, for FPGA to trigger events to STM.

FPGA-CTI

The FPGA-CTI allows the FPGA to send and receive triggers from the debug system.

Table 10-3: FPGA-CTI Signal Description Table

The following table lists the signal descriptions between the FPGA-CTI and FPGA.

Signal	Description
h2f_cti_trig_in[8]	Trigger input from FPGA
h2f_cti_trig_in_ack[8]	ACK signal to FPGA
h2f_cti_trig_out[8]	Trigger output to FPGA
h2f_cti_trig_out_ack[8]	ACK signal from FPGA
h2f_cti_clk	Clock input from FPGA
h2f_cti_fpga_clk_en	Clock enable driven by FPGA
h2f_cti_asicctl[8]	Signal from FPGA

Related Information

[Arm Infocenter](#)

For more information about the cross-trigger interface

TPIU

Figure 10-6: Trace Clock and Trace Data Width

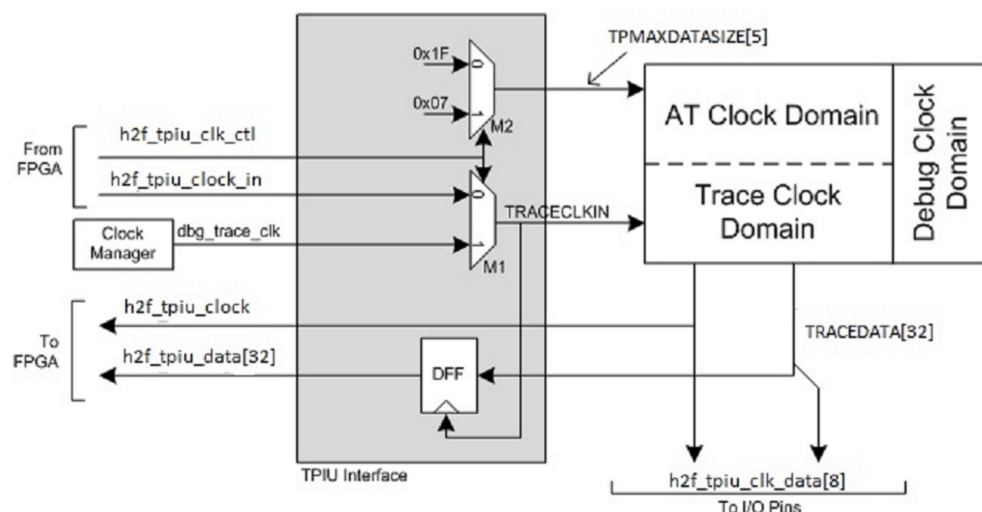


Table 10-4: TPIU Signals

The following table lists the signal descriptions between the TPIU and FPGA.

Signal	Description
<code>h2f_tpiu_clk_ctl</code>	Selects whether trace data is captured using the internal TPIU clock, which is the <code>dbg_trace_clk</code> signal from the clock manager; or an external clock provided as an input to the TPIU from the FPGA. 0 - use <code>h2f_tpiu_clock_in</code> 1 - use internal clock Note: When the FPGA is powered down or not configured the TPIU uses the internal clock.
<code>h2f_tpiu_data[32]</code>	32 bit trace data bus to the FPGA. Trace data changes on both edges of <code>h2f_tpiu_clock</code>. Note: When the FPGA is powered down or not configured, the TPIU sends the lower 8-bits trace data to I/Os.
<code>h2f_tpiu_clock_in</code>	Clock from the FPGA used to capture trace data.

Signal	Description
h2f_tpiu_clock	Clock output from TPIU

Debug Clocks

Table 10-5: CoreSight Clocks

Arm Clock Name	Clock Source	HPS Clock Signal Name	Description
ATCLK	Clock manager	dbg_at_clk	Trace bus clock.
CTICKL (for csCTI)	Clock manager	dbg_at_clk	Cross trigger interface clock for csCTI. It can be synchronous or asynchronous to CTMCLK.
CTICKL (for FPGA-CTI)	FPGA fabric	fpga_cti_clk	Cross trigger interface clock for FPGA-CTI.
CTICKL (for CTI-0 and CTI-1)	Clock manager	mpu_clk	Cross trigger interface clock for CTI-0 and CTI-1. It can be synchronous or asynchronous to CTMCLK.
CTMCLK (for csCTM)	Clock manager	dbg_clk	Cross trigger matrix clock for csCTM. It can be synchronous or asynchronous to CTICKL.
CTMCLK (for CTM)	Clock manager	mpu_clk	Cross trigger matrix clock for CTM. It can be synchronous or asynchronous to CTICKL.
DAPCLK	Clock manager	dbg_clk	DAP internal clock. It must be equivalent to PCLKDBG. dbg_clk must be at least twice as fast as the JTAG clock.
PCLKDBG	Clock manager	dbg_clk	Debug APB (DAPB) clock.
HCLK	Clock manager	dbg_clk	Used by the AHB-Lite master inside the DAP. It is asynchronous to DAPCLK. In the HPS, the AHB-Lite port uses same clock as DAPCLK.

Arm Clock Name	Clock Source	HPS Clock Signal Name	Description
PCLKSYS	Clock manager	l4_mp_clk	Used by the APB slave port inside the DAP. It is asynchronous to DAPCLK.
SWCLKTCK	JTAG interface FPGA fabric Note: There are two clock sources.	dap_tck tpiu_traceclk Note: There are two signal names.	The SWJ-DP clock driven by the external debugger through either the JTAG interface or the FPGA fabric. It is asynchronous to DAPCLK. When through the JTAG interface, this clock is the same as TCK of the JTAG interface.
TRACECLKIN	Clock manager	dbg_trace_clk	TPIU trace clock input. It is asynchronous to ATCLK. In the HPS, this clock can come from the clock manager or the FPGA fabric.

Related Information

[Arm Infocenter](#)

For more information about the CoreSight port names, refer to the *CoreSight Technology System Design Guide*.

Debug Resets

The CoreSight system uses several resets.

Table 10-6: CoreSight Resets

Arm Reset Name	Clock Source	HPS Reset Signal Name	Description
ATRESETn	Reset manager	dbg_rst_n	Trace bus reset. It resets all registers in the ATCLK domain.
nCTIRESET	Reset manager	dbg_rst_n	CTI reset signal. It resets all registers in the CTICLK domain. In the HPS, there are four instances of CTI. All four use the same reset signal.
DAPRESETn	Reset manager	dbg_rst_n	DAP internal reset. It is connected to PRESETDBGn.

Arm Reset Name	Clock Source	HPS Reset Signal Name	Description
PRESETDBGn	Reset manager	dbg_rst_n	Debug APB reset. Resets all registers clocked by PCLKDBG.
HRESETn	Reset manager	sys_dbg_rst_n	SoC-provided reset signal that resets all of the AMBA on-chip interconnect. Use this signal to reset the DAP AHB-Lite master port.
PRESETSYSn	Reset manager	sys_dbg_rst_n	Resets system APB slave port of DAP.
nCTMRESET	Reset manager	dbg_rst_n	CTM reset signal. It resets all signals clocked by CTMCLK.
nPOTRST	Reset manager	tap_cold_rst_n	True power on reset signal to the DAP SWJ-DP. It must only reset at power-on.
nTRST	JTAG interface	nTRST pin	Resets the DAP TAP controller inside the SWJ-DP. This signal is driven by the host using the JTAG connector.
TRESETn	Reset manager	dbg_rst_n	Reset signal for TPIU. Resets all registers in the TRACECLKIN domain.

The ETR stall enable field (`etrstallen`) of the `ctrl` register in the reset manager controls whether the ETR is requested to stall its AXI master interface to the L3 interconnect before a warm or debug reset.

The level 4 (L4) watchdog timers can be paused during debugging to prevent reset while the processor is stopped at a breakpoint.

Related Information

- [Reset Manager](#) on page 4-1
- [Watchdog Timer](#) on page 24-1
- [Arm Infocenter](#)

For more information about the CoreSight port names, refer to the *CoreSight Technology System Design Guide*.

CoreSight Debug and Trace Programming Model

This section describes programming model details specific to the device implementation of the Arm CoreSight technology.

The debug components can be configured to cause triggers when certain events occur. For example, soft logic in the FPGA fabric can signal an event which triggers an STM message injection into the trace stream.

Related Information

[Arm Infocenter](#)

Programming interface details of each CoreSight component.

CoreSight Component Address

CoreSight components are configured through memory-mapped registers, located at offsets relative to the CoreSight component base address. CoreSight component base addresses are accessible through the component address table in the DAP ROM.

Table 10-7: Coresight Component Address Table

The following table is located in the ROM table portion of the DAP.

ROM Entry	Offset[30:12]	Description
0x0	0x00001	ETF Component Base Address
0x1	0x00002	CTI Component Base Address
0x2	0x00003	TPIU Component Base Address
0x3	0x00004	Trace Funnel Component Base Address
0x4	0x00005	STM Component Base Address
0x5	0x00006	ETR Component Base Address
0x6	0x00007	FPGA-CTI Component Base Address
0x7	0x00100	A9 ROM
0x8	0x00080	FPGA ROM
0x9	0x00000	End of ROM

A host debugger can access this table at 0x80000000 through the DAP. HPS masters can access this ROM at 0xFF000000. Registers for a particular CoreSight component are accessed by adding the register offset to the CoreSight component base address, and adding that total to the base address of the ROM table.

The base address of the ROM table is different when accessed from the debugger (at 0x8000_0000) than when accessed from any HPS master (at 0xFF000000). For example, the CTI output enable (CTIOUTEN) register, CTIOUTEN[2] at offset 0xA8, can be accessed by the host debugger at 0x800020A8. To derive that value, add the host debugger access address to the ROM table of 0x80000000, to the CTI component base address of 0x00002000, to the CTIOUTEN[2] register offset of 0xA8.



STM Channels

The STM AXI slave is connected to the MPU, DMA, and FPGA-to-HPS bridge masters. Each master has up to 65536 channels where each channel occupies 256 bytes of address space, for a total of 16 MB per master. The HPS address map allocates 48 MB of consecutive address space to the STM AXI slave port, divided in three 16 MB segments.

Table 10-8: STM AXI Slave Port Address Allocation

Segment	Start Address	End Address
0	0xFC000000	0xFCFFFFFF
1	0xFD000000	0xFDFFFFFF
2	0xFE000000	0xFEFFFFFF

Each of the three masters can access any one of the three address segments. Your software design determines which master uses which segment, based on the value of bits 24 and 25 in the write address, AWADDRS[25:24]. Software must restrict each master to use only one of the three segments.

Table 10-9: STM AXI Address Fields

STM receives trace data over an AXI slave port. This table contains a list of signals associated with this interface.

AXI Signal Fields	Description
AWADDRS[7:0]	These bits index the 256 bytes of the stimulus port.
AWADDRS[23:8]	These bits identify the 65536 stimulus ports associated with a master.
AWADDRS[25:24]	These bits identify the three masters. Only 0, 1, and 2 are valid values.
AWADDRS[31:26]	Always 0x3F. Bits 24 to 31 combine to access 0xFC000000 through 0xFEFFFFFF.

Each STM message contains a master ID that tells the host debugger which master is associated with the message. The STM master ID is determined by combining a portion of the AWADDRS signal and the AWPROT protection bit. In the following table, MasterID[6] identifies the

Table 10-10: STM Master ID Calculation

Master ID Bits	AXI Signal Bits	Notes
Master ID[5:0]	AWADDRS[29:24]	The lowest two bits are sufficient to determine which master, but CoreSight uses a six-bit master ID.
Master ID[6]	AWPROT[1]	0 indicates a secure master; 1 indicates a nonsecure master.

In addition to access through STM channels, the higher-order 28 (31:4) of the 32 event signals are attached to the FPGA through the FPGA-CTI. These event signals allow the FPGA fabric to send additional messages using the STM.

Related Information

- [HPS-FPGA Bridges](#) on page 9-1
- [Arm Infocenter](#)
For more information, refer to "System Trace Macrocell" in the *Programmers' Model Architecture Specification*.
- [FPGA-CTI](#) on page 10-12

CTI Trigger Connections to Outside the Debug System

The following CTIs in the HPS debug system connect to outside the debug system:

- csCTI
- FPGA-CTI

csCTI

This section lists the trigger input, output, and output acknowledge pin connections implemented for csCTI in the debug system. The trigger input acknowledge signals are not connected to pins.

Table 10-11: csCTI Trigger Input Signals

The following table lists the trigger input pin connections implemented for csCTI.

Pin Number	Signal	Source
7	ASYNCOUT	STM
6	TRIGOUTHETE	STM
5	TRIGOUTSW	STM
4	TRIGOUTSPTE	STM
3	ACQCOMP	ETR
2	FULL	ETR

Pin Number	Signal	Source
1	ACQCOMP	ETF
0	FULL	ETF

Table 10-12: csCTI Trigger Output Signals

The following table lists the trigger output pin connections implemented for csCTI.

Pin Number	Signal	Destination
7	TRIGIN	ETF
6	FLUSHIN	ETF
5	HWEVENTS[3:2]	STM
4	HWEVENTS[1:0]	STM
3	TRIGIN	TPIU
2	FLUSHIN	TPIU
1	TRIGIN	ETR
0	FLUSHIN	ETR

Table 10-13: csCTI Trigger Output Acknowledge Signals

The following table lists the trigger output pin acknowledge connections implemented for csCTI.

Pin Number	Signal	Source
7	0	—
6	0	—
5	0	—
4	0	—
3	TRIGINACK	TPIU
2	FLUSHINACK	TPIU
1	0	—
0	0	—

FPGA-CTI

FPGA-CTI connects the debug system to the FPGA fabric. FPGA-CTI has all of its triggers available to the FPGA fabric.

Related Information

[Configuring Embedded Cross-Trigger Connections](#) on page 10-21

For more information about the triggers, refer to the "Configuring Embedded Cross-Trigger Connections" chapter.

Configuring Embedded Cross-Trigger Connections

CTI interfaces are programmable through a memory-mapped register interface.

The specific registers are described in the *CoreSight Components Technical Reference Manual*, which you can download from the Arm Infocenter.

To access registers in any CoreSight component through the debugger, the register offsets must be added to the CoreSight component's base address. That combined value must then be added to the address at which the ROM table is visible to the debugger (0x80000000).

Each CTI has two interfaces, the trigger interface and the channel interface. The trigger interface is the interface between the CTI and other components. It has eight trigger signals, which are hardwired to other components. The channel interface is the interface between a CTI and its CTM, with four bidirectional channels. The mapping of trigger interface to channel interface (and vice versa) in a CTI is dynamically configured. You can enable or disable each CTI trigger output and CTI trigger input connection individually.

For example, you can configure trigger input 0 in the FPGA-CTI to route to channel 3, and configure trigger output 3 in the FPGA-CTI and trigger output 7 in CTI-0 in the MPU debug subsystem to route from channel 3. This configuration causes a trigger at trigger input 0 in FPGA-CTI to propagate to trigger output 3 in the FPGA-CTI and trigger output 7 in CTI-0. Propagation can be single-to-single, single-to-multiple, multiple-to-single, and multiple-to-multiple.

A particular soft logic signal in the FPGA connected to a trigger input in the FPGA-CTI can be configured to trigger a flush of trace data to the TPIU. For example, you can configure channel 0 to trigger output 2 in the csCTI. Then configure trigger input T3 to channel 0 in FPGA-CTI. Trace data is flushed to the TPIU when a trigger is received at trigger output 2 in the csCTI.

Another soft logic signal in the FPGA connected to trigger input T2 in FPGA-CTI can be configured to trigger an STM message. The csCTI output triggers 4 and 5 are wired to the STM CoreSight component in the HPS. For example, configure channel 1 to trigger output 4 in the csCTI. Then configure trigger input T2 to channel 1 in FPGA-CTI.

Another soft logic signal in the FPGA fabric connected to trigger input T1 in FPGA-CTI can be configured to trigger a breakpoint on CPU 1. Trigger output 1 in CTI-1 is wired to the debug request (EDBGRQ) signal of CPU-1. For example, configure channel 2 to trigger output 1 in CTI-1. Then configure trigger input T1 to channel 2 in FPGA-CTI.

Related Information

- [Coresight Component Address](#) on page 10-17
- [Arm Infocenter](#)
For more information about the cross-trigger interface

Configuring Trigger Input 0

For example, you can configure trigger input 0 in the FPGA-CTI to route to channel 3, and configure trigger output 3 in the FPGA-CTI and trigger output 7 in CTI-0 in the MPU debug subsystem to route from channel 3. This configuration causes a trigger at trigger input 0 in FPGA-CTI to propagate to trigger output 3 in the FPGA-CTI and trigger output 7 in CTI-0. Propagation can be single-to-single, single-to-multiple, multiple-to-single, or multiple-to-multiple.

Triggering a Flush of Trace Data to the TPIU

A particular soft logic signal in the FPGA connected to a trigger input in the FPGA-CTI can be configured to trigger a flush of trace data to the TPIU. For example, you can configure channel 0 to trigger output 2 in csCTI. Then configure trigger input T3 to channel 0 in FPGA-CTI. Trace data is flushed to the TPIU when a trigger is received at trigger output 2 in csCTI.

Triggering an STM message

Another soft logic signal in the FPGA connected to trigger input T2 in FPGA-CTI can be configured to trigger an STM message. csCTI output triggers 4 and 5 are wired to the STM CoreSight component in the HPS. For example, configure channel 1 to trigger output 4 in csCTI. Then configure trigger input T2 to channel 1 in FPGA-CTI.

Triggering a Breakpoint on CPU 1

Another soft logic signal in the FPGA fabric connected to trigger input T1 in FPGA-CTI can be configured to trigger a breakpoint on CPU 1. Trigger output 1 in CTI-1 is wired to the external debug request (EDBGRQ) signal of CPU-1. For example, configure channel 2 to trigger output 1 in CTI-1. Then configure trigger input T1 to channel 2 in FPGA-CTI.

CoreSight Debug and Trace Address Map and Register Definitions

The address map and register definitions for CoreSight Debug and Trace consist of the following regions:

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter.
- [Arria V Address Map and Register Definitions](#)
Web-based address map and register definitions

SDRAM Controller Subsystem 11

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) SDRAM controller subsystem provides efficient access to external SDRAM for the Arm Cortex-A9 microprocessor unit (MPU) subsystem, the level 3 (L3) interconnect, and the FPGA fabric.

The SDRAM controller provides an interface between the FPGA fabric and HPS. The interface accepts Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) and Avalon® Memory-Mapped (Avalon-MM) transactions, converts those commands to the correct commands for the SDRAM, and manages the details of the SDRAM access.

Features of the SDRAM Controller Subsystem

The SDRAM controller subsystem offers programming flexibility, port and bus configurability, error correction, and power management.

- Support for double data rate 2 (DDR2), DDR3, and low-power DDR2 (LPDDR2) SDRAM
- Flexible row and column addressing with the ability to support up to 4 Gb of memory per chip select
- Optional 8-bit integrated error correction code (ECC) for 16- and 32-bit data widths
- User-configurable memory width of 8, 16, 16+ECC, 32, 32+ECC
- User-configurable timing parameters
- Two chip selects (DDR2 and DDR3)
- Command reordering (look-ahead bank management)
- Data reordering (out of order transactions)
- User-controllable bank policy on a per port basis for either closed page or conditional open page accesses
- User-configurable priority support with both absolute and weighted round-robin scheduling
- Flexible FPGA fabric interface with up to 6 ports that can be combined for a data width up to 256 bits using Avalon-MM and AXI interfaces
- Power management supporting self refresh, partial array self refresh (PASR), power down, and LPDDR2 deep power down

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

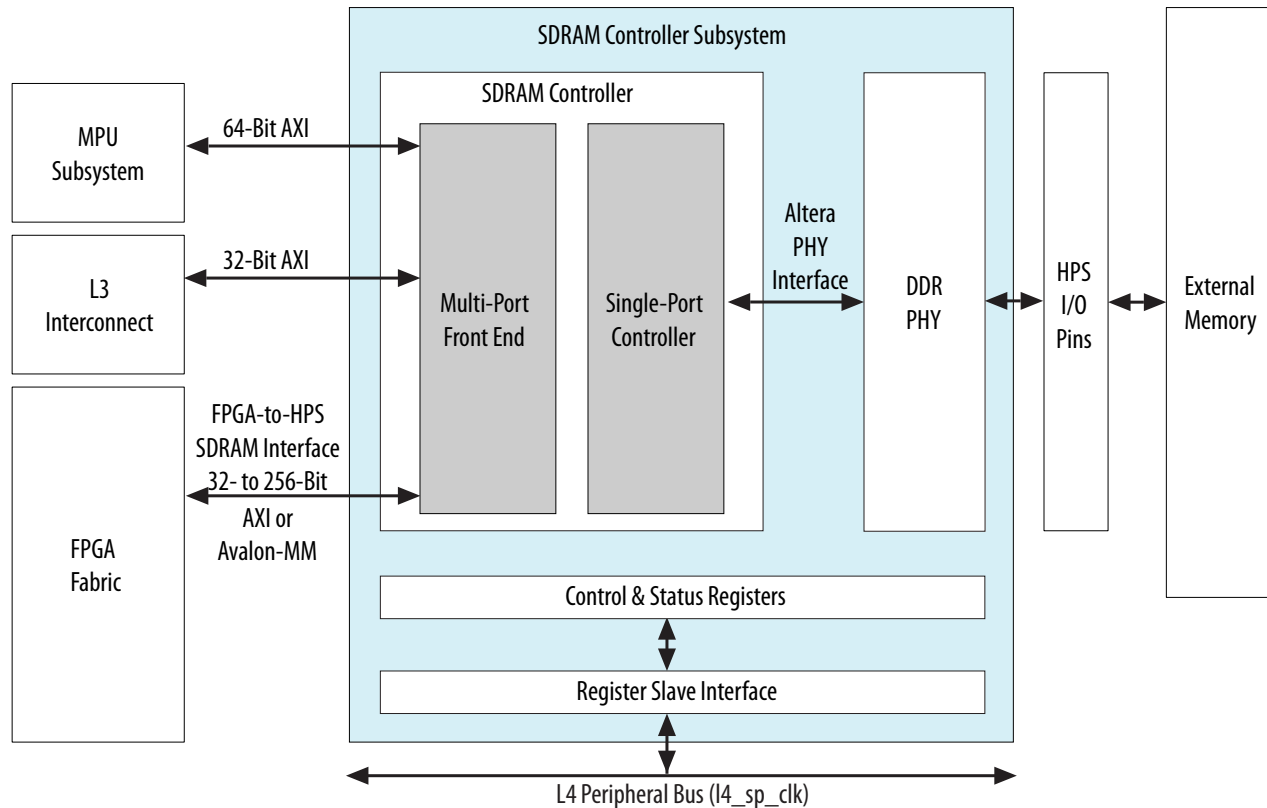
ISO
9001:2015
Registered

ALTERA
now part of Intel

SDRAM Controller Subsystem Block Diagram

The SDRAM controller subsystem connects to the MPU subsystem, the L3 interconnect, and the FPGA fabric. The memory interface consists of the SDRAM controller, the physical layer (PHY), control and status registers (CSRs), and their associated interfaces.

Figure 11-1: SDRAM Controller Subsystem High-Level Block Diagram



SDRAM Controller

The SDRAM controller provides high performance data access and run-time programmability. The controller reorders data to reduce row conflicts and bus turn-around time by grouping read and write transactions together, allowing for efficient traffic patterns and reduced latency.

The SDRAM controller consists of a multiport front end (MPFE) and a single-port controller. The MPFE provides multiple independent interfaces to the single-port controller. The single-port controller communicates with and manages each external memory device.

The MPFE FPGA-to-HPS SDRAM interface port has an asynchronous FIFO buffer followed by a synchronous FIFO buffer. Both the asynchronous and synchronous FIFO buffers have a read and write data FIFO depth of 8, and a command FIFO depth of 4. The MPU subsystem 64-bit AXI port and L3 interconnect 32-bit AXI port have asynchronous FIFO buffers with read and write data FIFO depth of 8, and command FIFO depth of 4.

DDR PHY

The DDR PHY provides a physical layer interface for read and write memory operations between the memory controller and memory devices. The DDR PHY has dataflow components, control components, and calibration logic that handle the calibration for the SDRAM interface timing.

Related Information

[Memory Controller Architecture](#) on page 11-6

SDRAM Controller Memory Options

Bank selects, and row and column address lines can be configured to work with SDRAMs of various technology and density combinations.

Table 11-1: SDRAM Controller Interface Memory Options

Memory Type (22)	Mbits	Column Address Bit Width	Bank Select Bit Width	Row Address Bit Width	Page Size	MBytes
DDR2	256	10	2	13	1024	32
	512	10	2	14	1024	64
	1024 (1 Gb)	10	3	14	1024	128
	2048 (2 Gb)	10	3	15	1024	256
	4096 (4 Gb)	10	3	16	1024	512
DDR3	512	10	3	13	1024	64
	1024 (1 Gb)	10	3	14	1024	128
	2048 (2 Gb)	10	3	15	1024	256
	4096 (4 Gb)	10	3	16	1024	512

⁽²²⁾ For all memory types shown in this table, the DQ width is 8.

Memory Type (22)	Mbits	Column Address Bit Width	Bank Select Bit Width	Row Address Bit Width	Page Size	MBytes
LPDDR2	64	9	2	12	512	8
	128	10	2	12	1024	16
	256	10	2	13	1024	32
	512	11	2	13	2048	64
	1024 (1 Gb) - S2 ⁽²³⁾	11	2	14	2048	128
	1024 (1 Gb) - S4 ⁽²⁴⁾	11	3	13	2048	128
	2048 (2 Gb) - S2 ⁽²³⁾	11	2	15	2048	256
	2048 (2 Gb) - S4 ⁽²⁴⁾	11	3	14	2048	256
	4096 (4 Gb)	12	3	14	4096	512

SDRAM Controller Subsystem Interfaces

MPU Subsystem Interface

The SDRAM controller connects to the MPU subsystem with a dedicated 64-bit AXI interface, operating on the mpu_l2_ram_clk clock domain.

L3 Interconnect Interface

The SDRAM controller interfaces to the L3 interconnect with a dedicated 32-bit AXI interface, operating on the l3_main_clk clock domain.

Related Information

[System Interconnect](#) on page 8-1

⁽²²⁾ For all memory types shown in this table, the DQ width is 8.

⁽²³⁾ S2 signifies a 2n prefetch size

⁽²⁴⁾ S4 signifies a 4n prefetch size

CSR Interface

The CSR interface connects to the level 4 (L4) bus and operates on the `l4_sp_clk` clock domain. The MPU subsystem uses the CSR interface to configure the controller and PHY, for example setting the memory timing parameter values or placing the memory in a low power state. The CSR interface also provides access to the status registers in the controller and PHY.

FPGA-to-HPS SDRAM Interface

The FPGA-to-HPS SDRAM interface provides masters implemented in the FPGA fabric access to the SDRAM controller subsystem in the HPS. The interface has three port types that are used to construct the following AXI or Avalon-MM interfaces:

- Command ports—issue read and write commands, and for receive write acknowledge responses
- 64-bit read data ports—receive data returned from a memory read
- 64-bit write data ports—transmit write data

The FPGA-to-HPS SDRAM interface supports six command ports, allowing up to six Avalon-MM interfaces or three AXI interfaces. Each command port can be used to implement either a read or write command port for AXI, or be used as part of an Avalon-MM interface. The AXI and Avalon-MM interfaces can be configured to support 32-, 64-, 128-, and 256-bit data.

Table 11-2: FPGA-to-HPS SDRAM Controller Port Types

Port Type	Available Number of Ports
Command	6
64-bit read data	4
64-bit write data	4

The FPGA-to-HPS SDRAM controller interface can be configured with the following characteristics:

- Avalon-MM interfaces and AXI interfaces can be mixed and matched as required by the fabric logic, within the bounds of the number of ports provided to the fabric.
- Because the AXI protocol allows simultaneous read and write commands to be issued, two SDRAM control ports are required to form an AXI interface.
- Because the data ports are natively 64-bit, they must be combined if wider data paths are required for the interface.
- Each Avalon-MM or AXI interface of the FPGA-to-HPS SDRAM interface operates on an independent clock domain.
- The FPGA-to-HPS SDRAM interfaces are configured during FPGA configuration.

The following table shows the number of ports needed to configure different bus protocols, based on type and data width.

Table 11-3: FPGA-to-HPS SDRAM Port Utilization

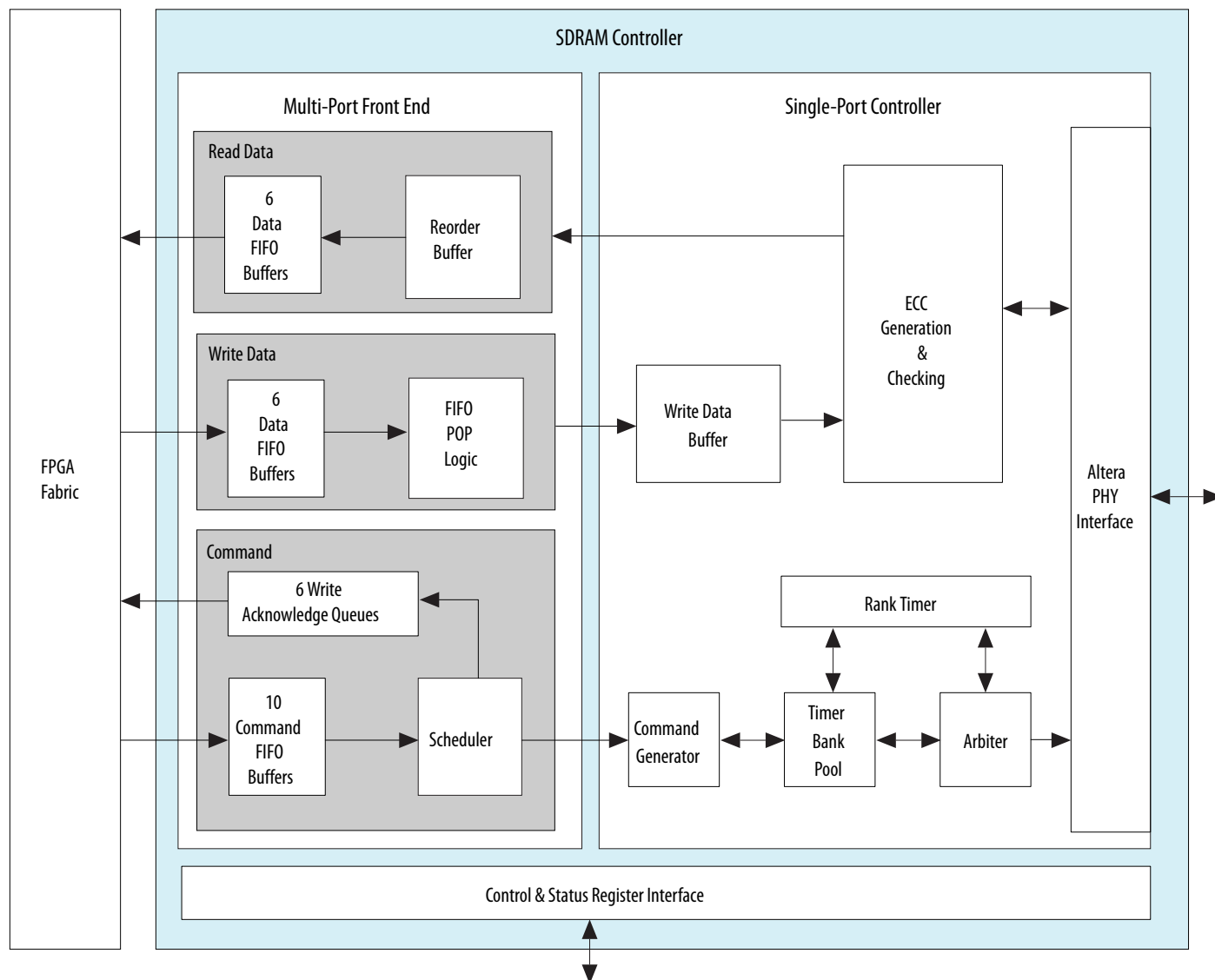
Bus Protocol	Command Ports	Read Data Ports	Write Data Ports
32- or 64-bit AXI	2	1	1

Bus Protocol	Command Ports	Read Data Ports	Write Data Ports
128-bit AXI	2	2	2
256-bit AXI	2	4	4
32- or 64-bit Avalon-MM	1	1	1
128-bit Avalon-MM	1	2	2
256-bit Avalon-MM	1	4	4
32- or 64-bit Avalon-MM write-only	1	0	1
128-bit Avalon-MM write-only	1	0	2
256-bit Avalon-MM write-only	1	0	4
32- or 64-bit Avalon-MM read-only	1	1	0
128-bit Avalon-MM read-only	1	2	0
256-bit Avalon-MM read-only	1	4	0

Memory Controller Architecture

The SDRAM controller consists of an MPFE, a single-port controller, and an interface to the CSRs.

Figure 11-2: SDRAM Controller Block Diagram



Multi-Port Front End

The Multi-Port Front End (MPFE) is responsible for scheduling pending transactions from the configured interfaces and sending the scheduled memory transactions to the single-port controller. The MPFE handles all functions related to individual ports.

The MPFE consists of three primary sub-blocks.

Command Block

The command block accepts read and write transactions from the FPGA fabric and the HPS. When the command FIFO buffer is full, the command block applies backpressure by deasserting the ready signal. For each pending transaction, the command block calculates the next SDRAM burst needed to progress on that transaction. The command block schedules pending SDRAM burst commands based on the user-supplied configuration, available write data, and unallocated read data space.

Write Data Block

The write data block transmits data to the single-port controller. The write data block maintains write data FIFO buffers and clock boundary crossing for the write data. The write data block informs the command block of the amount of pending write data for each transaction so that the command block can calculate eligibility for the next SDRAM write burst.

Read Data Block

The read data block receives data from the single-port controller. Depending on the port state, the read data block either buffers the data in its internal buffer or passes the data straight to the clock boundary crossing FIFO buffer. The read data block reorders out-of-order data for Avalon-MM ports.

In order to prevent the read FIFO buffer from overflowing, the read data block informs the command block of the available buffer area so the command block can pace read transaction dispatch.

Single-Port Controller

The single-port logic is responsible for following actions:

- Queuing the pending SDRAM bursts
- Choosing the most efficient burst to send next
- Keeping the SDRAM pipeline full
- Ensuring all SDRAM timing parameters are met

Transactions passed to the single-port logic for a single page in SDRAM are guaranteed to be executed in order, but transactions can be reordered between pages. Each SDRAM burst read or write is converted to the appropriate Altera PHY interface (AFI) command to open a bank on the correct row for the transaction (if required), execute the read or write command, and precharge the bank (if required).

The single-port logic implements command reordering (looking ahead at the command sequence to see which banks can be put into the correct state to allow a read or write command to be executed) and data reordering (allowing data transactions to be dispatched even if the data transactions are executed in an order different than they were received from the multi-port logic).

The single-port controller consists of eight sub-modules.

Command Generator

The command generator accepts commands from the MPFE and from the internal ECC logic, and provides those commands to the timer bank pool.

Related Information

[Memory Controller Architecture](#) on page 11-6

For more information, refer to the SDRAM Controller Block diagram.

Timer Bank Pool

The timer bank pool is a parallel queue that operates with the arbiter to enable data reordering. The timer bank pool tracks incoming requests, ensures that all timing requirements are met, and, on receiving write-data-ready notifications from the write data buffer, passes the requests to the arbiter.

Related Information

[Memory Controller Architecture](#) on page 11-6

For more information, refer to the SDRAM Controller Block diagram.

Arbiter

The arbiter determines the order in which requests are passed to the memory device. When the arbiter receives a single request, that request is passed immediately. When multiple requests are received, the arbiter uses arbitration rules to determine the order to pass requests to the memory device.

Related Information

[Memory Controller Architecture](#) on page 11-6

For more information, refer to the SDRAM Controller Block diagram.

Rank Timer

The rank timer performs the following functions:

- Maintains rank-specific timing information
- Ensures that only four activates occur within a specified timing window
- Manages the read-to-write and write-to-read bus turnaround time
- Manages the time-to-activate delay between different banks

Related Information

[Memory Controller Architecture](#) on page 11-6

For more information, refer to the SDRAM Controller Block diagram.

Write Data Buffer

The write data buffer receives write data from the MPFE and passes the data to the PHY, on approval of the write request.

Related Information

[Memory Controller Architecture](#) on page 11-6

For more information, refer to the SDRAM Controller Block diagram.

ECC Block

The ECC block consists of an encoder and a decoder-corrector, which can detect and correct single-bit errors, and detect double-bit errors. The ECC block can correct single-bit errors and detect double-bit errors resulting from noise or other impairments during data transmission.

Related Information

- [Memory Controller Architecture](#) on page 11-6
For more information, refer to the SDRAM Controller Block diagram.
- [External Memory Interfaces in Arria V Devices](#)

AFI Interface

The AFI interface provides communication between the controller and the PHY.

Related Information

[Memory Controller Architecture](#) on page 11-6

For more information, refer to the SDRAM Controller Block diagram.

CSR Interface

The CSR interface is accessible from the L4 bus. The interface allows code in the HPS MPU or soft IP cores in the FPGA fabric to configure and monitor the SDRAM controller.

Related Information

[Memory Controller Architecture](#) on page 11-6

For more information, refer to the SDRAM Controller Block diagram.

Functional Description of the SDRAM Controller Subsystem

MPFE Operation Ordering

Operation ordering is defined and enforced within a port, but not between ports. All transactions received on a single port for overlapping addresses execute in order. Requests arriving at different ports have no guaranteed order of service, except when a first transaction has completed before the second arrives.

Avalon-MM does not support write acknowledgement. When a port is configured to support Avalon-MM, you should read from the location that was previously written to ensure that the write operation has completed. When a port is configured to support AXI, the master accessing the port can safely issue a read operation to the same address as a write operation as soon as the write has been acknowledged. To keep write latency low, writes are acknowledged as soon as the transaction order is guaranteed—meaning that any operations received on any port to the same address as the write operation are executed after the write operation.

To reduce read latency, the single-port logic can return read data out of order to the multi-port logic. The returned data is rearranged to its initial order on a per port basis by the multi-port logic and no traffic reordering occurs between individual ports.

Read Data Handling

The MPFE contains a read buffer shared by all ports. If a port is capable of receiving returned data then the read buffer is bypassed. If the size of a read transaction is smaller than twice the memory interface width, the buffer RAM cannot be bypassed. The lowest memory latency occurs when the port is ready to receive data and the full width of the interface is utilized.

MPFE Multi-Port Arbitration

The HPS SDRAM controller multi-port front end (MPFE) contains a programmable arbiter. The arbiter decides which MPFE port gains access to the single-port memory controller.

The SDRAM transaction size that is arbitrated is a burst of two beats. This burst size ensures that the arbiter does not favor one port over another when the incoming transaction size is a large burst.

The arbiter makes decisions based on two criteria: priority and weight. The priority is an absolute arbitration priority where the higher priority ports always win arbitration over the lower priority ports. Because multiple ports can be set to the same priority, the weight value refines the port choice by implementing a round-robin arbitration among ports set to the same priority. This programmable weight allows you to assign a higher arbitration value to a port in comparison to others such that the highest weighted port receives more transaction bandwidth than the lower weighted ports of the same priority.

Before arbitration is performed, the MPFE buffers are checked for any incoming transactions. The priority of each port that has buffered transactions is compared and the highest priority wins. If multiple ports are of the same highest priority value, the port weight is applied to determine which port wins. Because the arbiter only allows SDRAM-sized bursts into the single-port memory controller, large transactions may

need to be serviced multiple times before the read or write command is fully accepted to the single-port memory controller. The MPFE supports dynamic tuning of the priority and weight settings for each port, with changes committed into the SDRAM controller at fixed intervals of time.

Arbitration settings are applied to each port of the MPFE. The memory controller supports a mix of Avalon-MM and AXI protocols. As defined in the "Port Mappings" section, the Avalon-MM ports consume a single command port while the AXI ports consume a pair of command ports to support simultaneous read and write transactions. In total, there are ten command ports for the MPFE to arbitrate. The following table illustrates the command port mapping within the HPS as well as the ports exposed to the FPGA fabric.

Table 11-4: HPS SDRAM MPFE Command Port Mapping

Command Port	Allowed Functions	Data Size
0, 2, 4	FPGA fabric AXI read command ports FPGA fabric Avalon-MM read or write command ports	32-bit to 256-bit data
1, 3, 5	FPGA fabric AXI write command ports FPGA fabric Avalon-MM read or write command ports	
6	L3 AXI read command port	32-bit data
7	MPU AXI read command port	64-bit data
8	L3 AXI write command port	32-bit data
9	MPU AXI write command port	64-bit data

When the FPGA ports are configured for AXI, the command ports are always assigned in groups of two starting with even number ports 0, 2, or 4 assigned to the read command channel. For example, if you configure the first FPGA-to-SDRAM port as AXI and the second port as Avalon-MM, you can expect the following mapping:

- Command port 0 = AXI read
- Command port 1 = AXI write
- Command port 2 = Avalon-MM read and write

Setting the MPFE Priority

The priority of each of the ten command ports is configured through the `userpriority` field of the `mppriority` register. This 30-bit register uses 3 bits per port to configure the priority. The lowest priority is 0x0 and the highest priority is 0x7. The bits are mapped in ascending order with bits [2:0] assigned to command port 0 and bits [29:27] assigned to command port 9.

Setting the MPFE Static Weights

The static weight settings used in the round-robin command port priority scheme are programmed in a 128-bit field distributed among four 32-bit registers:

- mpweight_0_4
- mpweight_1_4
- mpweight_2_4
- mpweight_3_4

Each port is assigned a 5-bit value within the 128-bit field, such that port 0 is assigned to bits [4:0] of the mpweight_0_4 register, port 1 is assigned to bits [9:5] of the mpweight_0_4 register up to port 9, which is assigned to bits [49:45] contained in the mpweight_1_4 register. The valid weight range for each port is 0x0 to 0x1F, with larger static weights representing a larger arbitration share.

Bits[113:50] in the mpweight_1_4, mpweight_2_4 and mpweight_3_4 registers, hold the sum of weights for each priority. This 64-bit field is divided into eight fields of 8-bits, each representing the sum of static weights. Bits[113:50] are mapped in ascending order with bits [57:50] holding the sum of static weights for all ports with priority setting 0x0, and bits [113:106] holding the sum of static weights for all ports with priority setting 0x7.

Example Using MPFE Priority and Weights

In this example, the following settings apply:

- FPGA MPFE ports 0 is assigned to AXI read commands and port 1 is assigned to AXI write commands.
- FPGA MPFE port 2 is assigned to Avalon-MM read and write commands.
- L3 master ports (ports 6 and 8) and the MPU ports (port 7 and 9) are given the lowest priority but the MPU ports are configured with more arbitration static weight than the L3 master ports.
- The FPGA MPFE command ports are given the highest priority; however, AXI ports 0 and 1 are given a larger static weight because they carry the highest priority traffic in the entire system. Assigning a high priority and larger static weight ensures ports 0 and 1 receives the highest quality-of-service (QoS).

The table below details the port weights and sum of weights.

Table 11-5: SDRAM MPFE Port Priority, Weights and Sum of Weights

Priority	Weights										Sum of Weights
-	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7	Port 8	Port 9	-
1	10	10	5	0	0	0	0	0	0	0	25
0	0	0	0	0	0	0	1	4	1	4	10

If the FPGA-to-SDRAM ports are configured according to the table and if both ports are accessed concurrently, you can expect the AXI port to receive 80% of the total service. This value is determined by taking the sum of port 0 and 1 weights divided by the total weight for all ports of priority 1. The remaining 20% of bandwidth is allocated to the Avalon-MM port. With these port settings, any FPGA transaction buffered by the MPFE for either slave port blocks the MPU and L3 masters from having their buffered

transactions serviced. To avoid transaction starvation, you should assign ports the same priority level and adjust the bandwidth allocated to each port by adjusting the static weights.

MPFE Weight Calculation

The MPFE applies a deficit round-robin arbitration scheme to determine which port is serviced. The larger the port weight, the more often it is serviced. Ports are serviced only when they have buffered transactions and are set to the highest priority of all the ports that also have buffered transactions. The arbiter determines which port to service by examining the running weight of all the same ports at the same priority level and the largest running weight is serviced.

Each time a port is drained of all transactions, its running weight is set to 0x80. Each time a port is serviced, the static weight is added and the sum of weights is subtracted from the running weight for that port. Each time a port is not serviced (same priority as another port but has a lower running weight), the static weight for the port is added to the running weight of the port for that particular priority level. The running weight additions and subtractions are only applied to one priority level, so any time ports of a different priority level are being serviced, the running weight of a lower priority port is not modified.

MPFE Multi-Port Arbitration Considerations for Use

When using MPFE multi-port arbitration, the following considerations apply:

- To ensure that the dynamic weight value does not roll over when a port is serviced, the following equation should always be true:

$$(\text{sum of weights} - \text{static weight}) < 128$$

If the running weight remains less than 128, arbitration for that port remains functional.

- The memory controller commits the priority and weight registers into the MPFE arbiter once every 10 SDRAM clock cycles. As a result, when the `mppriority` register and `mpweight_*_4` registers are configured at run time, the update interval can occur while the registers are still being written and ports can have different priority or weights than intended for a brief period. Because the `mppriority` and `mpweight_*_4` registers can be updated in a single 32-bit transaction, Intel recommends updating first to ensure that transactions that need to be serviced have the appropriate priority after the next update. Because the static weights are divided among four 32-bit registers
- In addition to the `mppriority` register and `mpweight_*_*` registers, the `remappriority` register adds another level of priority to the port scheduling. By programming bit N in the `priorityremap` field of the `remappriority` register, any port with an absolute priority N is sent to the front of the single port command queue and is serviced before any other transaction. Please refer to the `remappriority` register for more details.

The scheduler is work-conserving. Write operations can only be scheduled when enough data for the SDRAM burst has been received. Read operations can only be scheduled when sufficient internal memory is free and the port is not occupying too much of the read buffer.

Related Information

[Port Mappings](#) on page 11-26

Refer to the "Port Mappings" section, for more information about command, read and write port assignments.

MPFE SDRAM Burst Scheduling

SDRAM burst scheduling recognizes addresses that access the same row/bank combination, known as open page accesses. Operations to a page are served in the order in which they are received by the single-port controller. Selection of SDRAM operations is a two-stage process. First, each pending transaction

must wait for its timers to be eligible for execution. Next, the transaction arbitrates against other transactions that are also eligible for execution.

The following rules govern transaction arbitration:

- High-priority operations take precedence over lower-priority operations
- If multiple operations are in arbitration, read operations have precedence over write operations
- If multiple operations still exist, the oldest is served first

A high-priority transaction in the SDRAM burst scheduler wins arbitration for that bank immediately if the bank is idle and the high-priority transaction's chip select, row, or column fields of the address do not match an address already in the single-port controller. If the bank is not idle, other operations to that bank yield until the high-priority operation is finished. If the chip select, row, and column fields match an earlier transaction, the high-priority transaction yields until the earlier transaction is completed.

Clocking

The FPGA fabric ports of the MPFE can be clocked at different frequencies. Synchronization is maintained by clock-domain crossing logic in the MPFE. Command ports can operate on different clock domains, but the data ports associated with a given command port must be attached to the same clock as that command port.

Note: A command port paired with a read and write port to form an Avalon-MM interface must operate at the same clock frequency as the data ports associated with it.

Single-Port Controller Operation

The single-port controller increases the performance of memory transactions through command and data re-ordering, enforcing bank policies, combining write operations and allowing burst transfers. Correction of single-bit errors and detection of double-bit errors is handled in the ECC module of the single-port Controller.

SDRAM Interface

The SDRAM interface is up to 40 bits wide and can accommodate 8-bit, 16-bit, 16-bit plus ECC, 32-bit, or 32-bit plus ECC configurations. The SDRAM interface supports LPDDR2, DDR2, and DDR3 memory protocols.

Command and Data Reordering

The heart of the SDRAM controller is a command and data reordering engine. Command reordering allows banks for future transactions to be opened before the current transaction finishes.

Data reordering allows transactions to be serviced in a different order than they were received when that new order allows for improved utilization of the SDRAM bandwidth. Operations to the same bank and row are performed in order to ensure that operations which impact the same address preserve the data integrity.

The following figure shows the relative timing for a write/read/write/read command sequence performed in order and then the same command sequence performed with data reordering. Data reordering allows the write and read operations to occur in bursts, without bus turnaround timing delay or bank reassignment.

Figure 11-3: Data Reordering Effect

Data Reordering Off

Command Address	WR B0R0					RD B1R0				WR B0R0					RD B1R0

Data Reordering On

Command Address	WR B0R0		WR B0R0				RD B1R0		RD B1R0						

The SDRAM controller schedules among all pending row and column commands every clock cycle.

Bank Policy

The bank policy of the SDRAM controller allows you to request that a transaction bank remain open after an operation has finished so that future accesses do not delay in activating the same bank and row combination. The controller supports only eight simultaneously-opened banks, so an open bank might get closed if the bank resource is needed for other operations.

Open bank resources are allocated dynamically as SDRAM burst transactions are scheduled. Bank allocation is requested automatically by the controller when an incoming transaction spans multiple SDRAM bursts or by the extended command interface. When a bank must be reallocated, the least-recently-used open bank is used as the replacement.

If the controller determines that the next pending command causes the bank request to not be honored, the bank might be held open or closed depending on the pending operation. A request to close a bank with a pending operation in the timer bank pool to the same row address causes the bank to remain open. A request to leave a bank open with a pending command to the same bank but a different row address causes a precharge operation to occur.

Write Combining

The SDRAM controller combines write operations from successive bursts on a port where the starting address of the second burst is one greater than the ending address of the first burst and the resulting burst length does not overflow the 11-bit burst-length counters.

Write combining does not occur if the previous bus command has finished execution before the new command has been received.

Burst Length Support

The controller supports burst lengths of 2, 4, 8, and 16. Data widths of 8, 16, and 32 bits are supported for non-ECC operation and data widths of 24 and 40 bits are supported for operations with ECC enabled. The following table shows the type of SDRAM for each burst length.

Table 11-6: SDRAM Burst Lengths

Burst Length	SDRAM
4	LPDDR2, DDR2
8	DDR2, DDR3, LPDDR2

Burst Length	SDRAM
16	LPDDR2

Width Matching

The SDRAM controller automatically performs data width conversion.

ECC

The single-port controller supports memory ECC calculated by the controller. The controller ECC employs standard Hamming logic to detect and correct single-bit errors and detect double-bit errors. The controller ECC is available for 16-bit and 32-bit widths, each requiring an additional 8 bits of memory, resulting in an actual memory width of 24-bits and 40-bits, respectively.

Functions the controller ECC provides are:

- Byte Writes
- ECC Write Backs
- Notification of ECC Errors

Byte Writes

The memory controller performs a read-modify-write operation to ensure that the ECC data remains valid when a subset of the bits of a word is being written.

Byte writes with ECC enabled are executed as a read-modify-write. Typical operations only use a single entry in the timer bank pool. Controller ECC enabled sub-word writes use two entries. The first operation is a read and the second operation is a write. These two operations are transferred to the timer bank pool with an address dependency so that the write cannot be performed until the read data has returned. This approach ensures that any subsequent operations to the same address (from the same port) are executed after the write operation, because they are ordered on the row list after the write operation.

If an entire word is being written (but less than a full burst) and the DM pins are connected, no read is necessary and only that word is updated. If controller ECC is disabled, byte-writes have no performance impact.

ECC Write Backs

If the controller ECC is enabled and a read operation results in a correctable ECC error, the controller corrects the location in memory, if write backs are enabled. The correction results in scheduling a new read-modify-write.

A new read is performed at the location to ensure that a write operation modifying the location is not overwritten. The actual ECC correction operation is performed as a read-modify-write operation. ECC write backs are enabled and disabled through the `cfg_enable_ecc_code_overwrites` field in the `ctrlcfg` register.

Note: Double-bit errors do not generate read-modify-write commands. Instead, double-bit error address and count are reported through the `erraddr` and `dbecount` registers, respectively. In addition, a double-bit error interrupt can be enabled through the `dramintr` register.

User Notification of ECC Errors

When an ECC error occurs, an interrupt signal notifies the MPU subsystem, and the ECC error information is stored in the status registers. The memory controller provides interrupts for single-bit and double-bit errors.

The status of interrupts and errors are recorded in status registers, as follows:

- The `dramsts` register records interrupt status.
- The `dramintr` register records interrupt masks.
- The `sbecount` register records the single-bit error count.
- The `dbecount` register records the double-bit error count.
- The `erraddr` register records the address of the most recent error.

For a 32-bit interface, ECC is calculated across a span of 8 bytes, meaning the error address is a multiple of 8 bytes (4-bytes*2 burst length). To find the byte address of the word that contains the error, you must multiply the value in the `erraddr` register by 8.

Related Information

Cortex-A9 Microprocessor Unit Subsystem

Information about ECC error interrupts

Interleaving Options

The controller supports the following address-interleaving options:

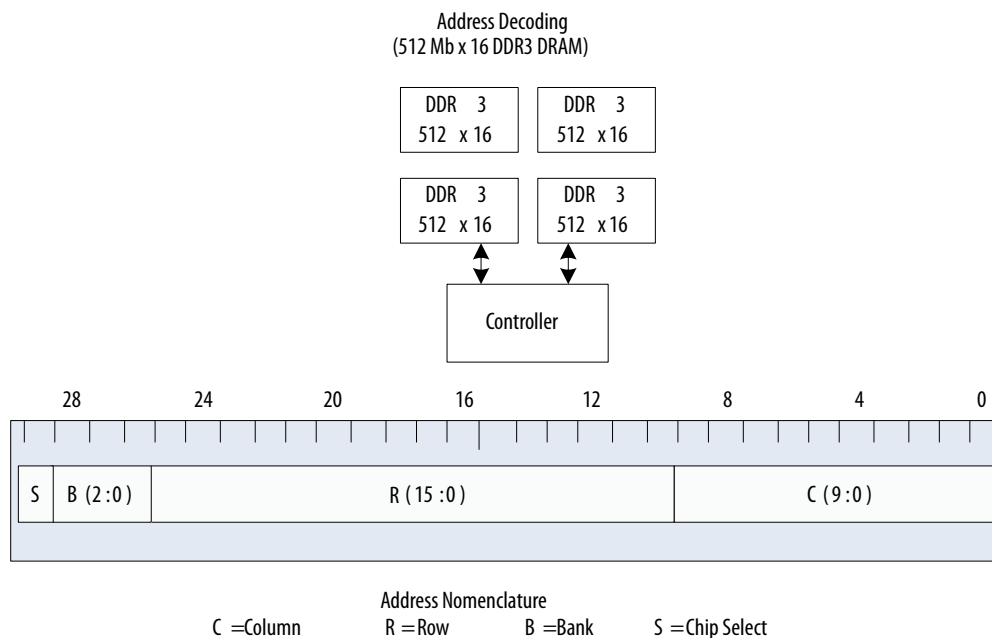
- Non-interleaved
- Bank interleave without chip select interleave
- Bank interleave with chip select interleave

The following interleaving examples use 512 megabits (Mb) x 16 DDR3 chips and are documented as byte addresses. For RAMs with smaller address fields, the order of the fields stays the same but the widths may change.

Non-interleaved

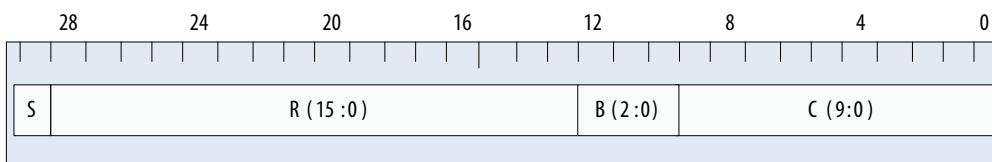
RAM mapping is non-interleaved.

Figure 11-4: Non-interleaved Address Decoding

**Bank Interleave Without Chip Select Interleave**

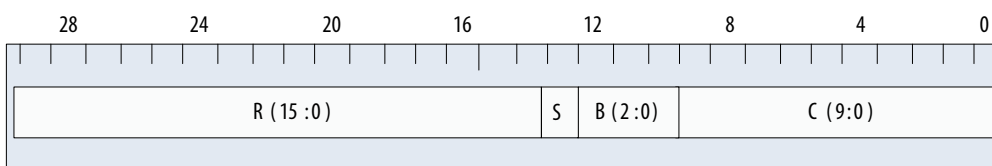
Bank interleave without chip select interleave swaps row and bank from the non-interleaved address mapping. This interleaving allows smaller data structures to spread across all banks in a chip.

Figure 11-5: Bank Interleave Without Chip Select Interleave Address Decoding

**Bank Interleave with Chip Select Interleave**

Bank interleave with chip select interleave moves the row address to the top, followed by chip select, then bank, and finally column address. This interleaving allows smaller data structures to spread across multiple banks and chips (giving access to 16 total banks for multithreaded access to blocks of memory). Memory timing is degraded when switching between chips.

Figure 11-6: Bank Interleave With Chip Select Interleave Address Decoding



AXI-Exclusive Support

The single-port controller supports AXI-exclusive operations. The controller implements a table shared across all masters, which can store up to 16 pending writes. Table entries are allocated on an exclusive read and table entries are deallocated on a successful write to the same address by any master.

Any exclusive write operation that is not present in the table returns an exclusive fail as acknowledgement to the operation. If the table is full when the exclusive read is performed, the table replaces a random entry.

Note: When using AXI-exclusive operations, accessing the same location from Avalon-MM interfaces can result in unpredictable results.

Memory Protection

The single-port controller has address protection to allow the software to configure basic protection of memory from all masters in the system. If the system has been designed exclusively with AMBA masters, TrustZone® is supported. Ports that use Avalon-MM can be configured for port level protection.

Memory protection is based on physical addresses in memory. The single-port controller can configure up to 20 rules to allow or prevent masters from accessing a range of memory based on their AxIDs, level of security and the memory region being accessed. If no rules are matched in an access, then default settings take effect.

The rules are stored in an internal protection table and can be accessed through indirect addressing offsets in the `protruledwr` register in the CSR. To read a specific rule, set the `readrule` bit and write the appropriate offset in the `ruleoffset` field of the `protruledwr` register.

To write a new rule, three registers in the CSR must be configured:

1. The `protportdefault` register is programmed to control the default behavior of memory accesses when no rules match. When a bit is clear, all default accesses from that port pass. When a bit is set, all default accesses from that port fails. The bits are assigned as follows:

Table 11-7: `protportdefault` register

Bits	Description
31:10	reserved
9	When this bit is set to 1, deny CPU writes during a default transaction. When this bit is clear, allow CPU writes during a default transaction.
8	When this bit is set to 1, deny L3 writes during a default transaction. When this bit is clear, allow L3 writes during a default transaction.
7	When this bit is set to 1, deny CPU reads during a default transaction. When this bit is clear, allow CPU reads during a default transaction.
6	When this bit is set to 1, deny L3 reads during a default transaction. When this bit is clear, allow L3 reads during a default transaction.

Bits	Description
5:0	When this bit is set to 1, deny accesses from FPGA-to-SDRAM ports 0 through 5 during a default transaction. When this bit is clear, allow accesses from FPGA-to-SDRAM ports 0 through 5 during a default transaction.

2. The `protruleid` register gives the bounds of the AxID value that allows an access
3. The `protruledata` register configures the specific security characteristics for a rule.

Once the registers are configured, they can be committed to the internal protection table by programming the `ruleoffset` field and setting the `writerule` bit in the `protruledwr` register.

Secure and non-secure regions are specified by rules containing a starting address and ending address with 1 MB boundaries for both addresses. You can override the port defaults and allow or disallow all transactions.

The following table lists the fields that you can specify for each rule.

Table 11-8: Fields for Rules in Memory Protection Table

Field	Width	Description
Valid	1	Set to 1 to activate the rule. Set to 0 to deactivate the rule.
Port Mask ⁽²⁵⁾	10	Specifies the set of ports to which the rule applies, with one bit representing each port, as follows: bits 0 to 5 correspond to FPGA fabric ports 0 to 5, bit 6 corresponds to AXI L3 interconnect read, bit 7 is the CPU read, bit 8 is L3 interconnect write, and bit 9 is the CPU write.
AxID_low ⁽²⁵⁾	12	Low transfer AxID of the rules to which this rule applies. Incoming transactions match if they are greater than or equal to this value. Ports with smaller AxIDs have the AxID shifted to the lower bits and zero padded at the top.
AxID_high ⁽²⁵⁾	12	High transfer AxID of the rules to which this rule applies. Incoming transactions match if they are less than or equal to this value.
Address_low	12	Points to a 1MB block and is the lower address. Incoming addresses match if they are greater than or equal to this value.
Address_high	12	Upper limit of address. Incoming addresses match if they are less than or equal to this value.

⁽²⁵⁾ Although AxID and Port Mask could be redundant, including both in the table allows possible compression of rules. If masters connected to a port do not have contiguous AxIDs, a port-based rule might be more efficient than an AxID-based rule, in terms of the number of rules needed.

Field	Width	Description
Protection	2	A value of 0x0 indicates that the rule applies to non-secure transactions; a value of 0x1 indicates the rule applies to non-secure transactions. Values 0x2 and 0x3 set the region to shared, meaning both secure and non-secure accesses are valid.
Fail/allow	1	Set this value to 1 to force the operation to fail or succeed.

Each port has a default access status of either allow or fail. Rules with the opposite allow/fail value can override the default. The system evaluates each transaction against every rule in the memory protection table. If a transaction arrives at a port that defaults to access allowed, it fails only if a rule with the fail bit matches the transaction. Conversely, if a transaction arrives at a port that has the default rule set to access denied, it allows access only if there is a matching rule that forces accessed allowed. Transactions that fail the protection rules return a slave error (SLVERR).

The recommended sequence for writing a rule is:

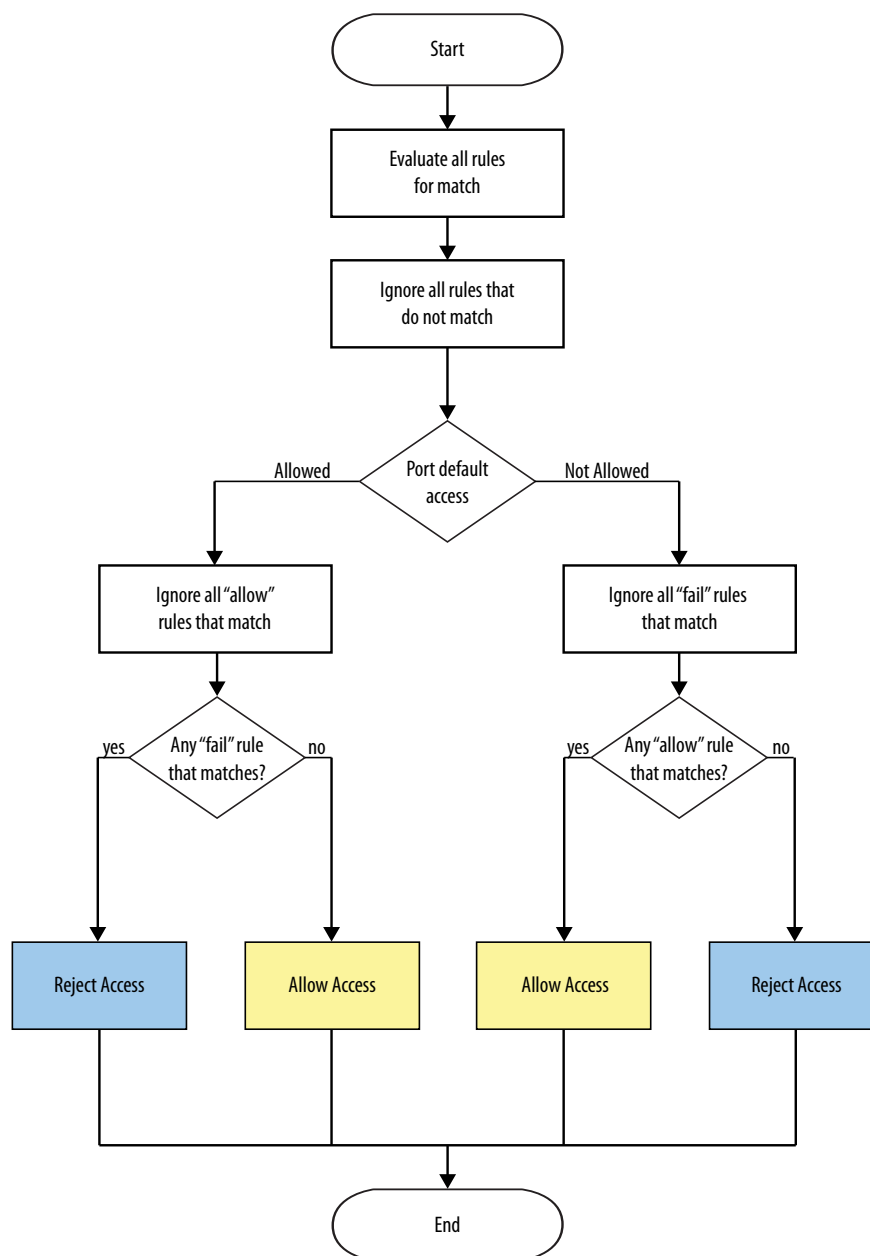
1. Write the `protruledwr` register fields as follows:
 - `ruleoffset` = offset selected by user that points to indirect offset in an internal protection table..
 - `writerule` = 0
 - `readrule` = 0
2. Write the `protruleaddr`, `protruleid`, and `protruledata` registers so you configure the rule you would like to enforce.
3. Write the `protruledwr` register fields as follows:
 - `ruleoffset` = offset of the rule that needs to be written
 - `writerule` = 1
 - `readrule` = 0

Similarly, the recommended sequence for reading a rule is:

1. Write the `protruledwr` register fields as follows:
 - `ruleoffset` = offset of the rule that needs to be written
 - `writerule` = 0
 - `readrule` = 0
2. Write the `protruledwr` register fields as follows:
 - `ruleoffset` = offset of the rule that needs to be read
 - `writerule` = 0
 - `readrule` = 1
3. Read the values of the `protruleaddr`, `protruleid`, and `protruledata` registers to determine the rule parameters.

The following figure represents an overview of how the protection rules are applied. There is no priority among the 20 rules. All rules are always evaluated in parallel.

Figure 11-7: SDRAM Protection Access Flow Diagram



Exclusive transactions are security checked on the read operation only. A write operation can occur only if a valid read is marked in the internal exclusive table. Consequently, a master performing an exclusive read followed by a write, can write to memory only if the exclusive read was successful.

Related Information

[Arm TrustZone®](#)

For more information about TrustZone® refer to the Arm web page.

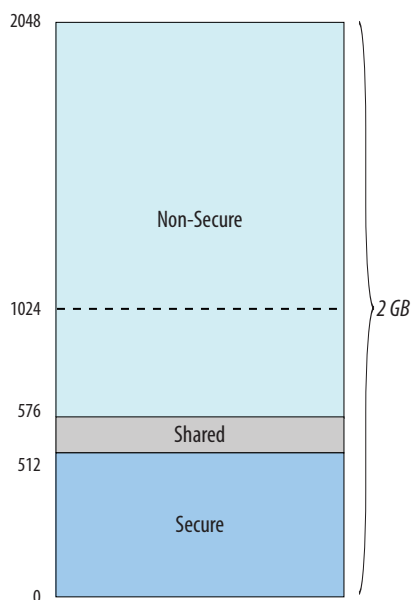
Example of Configuration for TrustZone

For a TrustZone configuration, memory is TrustZone divided into a range of memory accessible by secure masters and a range of memory accessible by non-secure masters. The two memory address ranges may have a range of memory that overlaps.

This example implements the following memory configuration:

- 2 GB total RAM size
- 0—512 MB dedicated secure area
- 513—576 MB shared area
- 577—2048 MB dedicated non-secure area

Figure 11-8: Example Memory Configuration



In this example, each port is configured by default to disallow all accesses. The following table shows the two rules programmed into the memory protection table.

Table 11-9: Rules in Memory Protection Table for Example Configuration

Rule #	Port Mask	AxID Low	AxID High	Address Low	Address High	protruledata.security	Fail/Allow
1	0x3FF (1023)	0x000	0xFFFF (4095)	0	576	0x1	Allow
2	0x3FF (1023)	0x000	0xFFFF (4095)	512	2047	0x0	Allow

The port mask value, AxID Low, and AxID High, apply to all ports and all transfers within those ports. Each access request is evaluated against the memory protection table, and fails unless there is a rule match allowing a transaction to complete successfully.

Table 11-10: Result for a Sample Set of Transactions

Operation	Source	Address Accesses	Security Access Type	Result	Comments
Read	CPU	4096	secure	Allow	Matches rule 1.
Write	CPU	536, 870, 912	secure	Allow	Matches rule 1.
Write	L3 attached masters	605, 028, 350	secure	Fail	Does not match rule 1 (out of range of the address field), does not match rule 2 (protection bit incorrect).
Read	L3 attached masters	4096	non-secure	Fail	Does not match rule 1 (AxPROT signal value wrong), does not match rule 2 (not in address range).
Write	CPU	536, 870, 912	non-secure	Allow	Matches rule 2.
Write	L3 attached masters	605, 028, 350	non-secure	Allow	Matches rule 2.

Note: If a master is using the Accelerator Coherency Port (ACP) to maintain cache coherency with the Cortex-A9 MPCore processor, then the address ranges in the rules of the memory protection table should be made mutually exclusive, such that the secure and non-secure regions do not overlap and any area that is shared is part of the non-secure region. This configuration prevents coherency issues from occurring.

SDRAM Power Management

The SDRAM controller subsystem supports the following power saving features in the SDRAM:

- Partial array self-refresh (PASR)
- Power down
- Deep power down for LPDDR2

To enable self-refresh for the memories of one or both chip selects, program the `selfshreq` bit and the `sefrrshmask` bit in the `lowpwreq` register.

Power-saving mode initiates either due to a user command or from inactivity. The number of idle clock cycles after which a memory can be put into power-down mode is programmed through the `autopdycycles` field of the `lowpwrtiming` register.

Power-down mode forces the SDRAM burst-scheduling bank-management logic to close all banks and issue the power down command. The SDRAM automatically reactivates when an active SDRAM command is received.

To enable deep power down request for the LPDDR2 memories of one or both chip selects, program the `deppwrnreq` bit and the `deppwrnmask` field of the `lowpwreq` register.

Other power-down modes are performed only under user control.

DDR PHY

The DDR PHY connects the memory controller and external memory devices in the speed critical command path.

The DDR PHY implements the following functions:

- Calibration—the DDR PHY supports the JEDEC-specified steps to synchronize the memory timing between the controller and the SDRAM chips. The calibration algorithm is implemented in software.
- Memory device initialization—the DDR PHY performs the mode register write operations to initialize the devices. The DDR PHY handles re-initialization after a deep power down.
- Single-data-rate to double-data-rate conversion.

DDR Calibration

The SDRAM Controller calibrates across multiple SDRAM banks. An entire row is calibrated at bank 0 and bank 7 in each rank. Thus, if you have a 4 Gb memory made up of two ranks, rank 0 is calibrated in banks 0 and 7, and rank 1 is calibrated in banks 0 and 7.

You can refer to the "Interleaving Options" section to identify which memory locations are affected by calibration.

Note: The SDRAM Controller does not preserve memory contents through a calibration cycle.

Related Information

[Interleaving Options](#) on page 11-17

Clocks

All clocks are assumed to be asynchronous with respect to the `ddr_dqs_clk` memory clock. All transactions are synchronized to memory clock domain.

Table 11-11: SDRAM Controller Subsystem Clock Domains

Clock Name	Description
<code>ddr_dq_clk</code>	Clock for PHY
<code>ddr_dqs_clk</code>	Clock for MPFE, single-port controller, CSR access, and PHY
<code>ddr_2x_dqs_clk</code>	Clock for PHY that provides up to 2 times <code>ddr_dq_clk</code> frequency
<code>l4_sp_clk</code>	Clock for CSR interface

Clock Name	Description
mpu_l2_ram_clk	Clock for MPU interface
l3_main_clk	Clock for L3 interface
f2h_sdram_clk[5:0]	Six separate clocks used for the FPGA-to-HPS SDRAM ports to the FPGA fabric

In terms of clock relationships, the FPGA fabric connects the appropriate clocks to write data, read data, and command ports for the constructed ports.

Related Information

[Clock Manager](#) on page 3-1

Resets

The SDRAM controller subsystem supports a full reset (cold reset) and a warm reset. The SDRAM controller can be configured to preserve memory contents during a warm reset.

To preserve memory contents, the reset manager can request that the single-port controller place the SDRAM in self-refresh mode prior to issuing the warm reset. If self-refresh mode is enabled before the warm reset to preserve memory contents, the PHY and the memory timing logic is not reset, but the rest of the controller is reset.

Related Information

[Reset Manager](#) on page 4-1

Taking the SDRAM Controller Subsystem Out of Reset

When a cold or warm reset is issued in the HPS, the Reset Manager resets this module and holds it in reset until software releases it.

After the MPU boots up, it can deassert the reset signal by clearing the appropriate bits in the Reset Manager's corresponding reset trigger.

Related Information

[Modules Requiring Software Deassert](#) on page 4-9

For more details about reset registers, refer to the Reset Manager.

Port Mappings

The memory interface controller has a set of command, read data, and write data ports that support AXI3, AXI4 and Avalon-MM. Tables are provided to identify port assignments and functions.

Table 11-12: Command Port Assignments

Command Port	Allowed Functions
0, 2, 4	FPGA fabric AXI read command ports FPGA fabric Avalon-MM read or write command ports
1, 3, 5	FPGA fabric AXI write command ports FPGA fabric Avalon-MM read or write command ports
6	L3 AXI read command port
7	MPU AXI read command port
8	L3 AXI write command port
9	MPU AXI write command port

Table 11-13: Read Port Assignments

Read Port	Allowed Functions
0, 1, 2, 3	64-bit read data from the FPGA fabric. When 128-bit data read ports are created, then read data ports 0 and 1 get paired as well as 2 and 3.
4	32-bit L3 read data port
5	64-bit MPU read data port

Table 11-14: Write Port Assignments

Write Port	Allowed Functions
0, 1, 2, 3	64-bit write data from the FPGA fabric. When 128-bit data write ports are created, then write data ports 0 and 1 get paired as well as 2 and 3.
4	32-bit L3 write data port
5	64-bit MPU write data port

Initialization

The SDRAM controller subsystem has control and status registers (CSRs) which control the operation of the controller including DRAM type, DRAM timing parameters and relative port priorities. It also has a small set of bits which depend on the FPGA fabric to configure ports between the memory controller and

the FPGA fabric; these bits are set for you when you configure your implementation using the HPS GUI in Platform Designer (Standard).

The CSRs are configured using a dedicated slave interface, which provides access to the registers. This region controls all SDRAM operation, MPFE scheduler configuration, and PHY calibration.

The FPGA fabric interface configuration is programmed into the FPGA fabric and the values of these register bits can be read by software. The ports can be configured without software developers needing to know how the FPGA-to-HPS SDRAM interface has been configured.

FPGA-to-SDRAM Protocol Details

The following topics summarize signals for the Avalon-MM Bidirectional port, Avalon-MM Write Port, Avalon-MM Read Port, and AXI port.

Note: If your device has multiple FPGA hardware images, then the same FPGA-to-SDRAM port configuration should be used across all designs.

Avalon-MM Bidirectional Port

The Avalon-MM bidirectional ports are standard Avalon-MM ports used to dispatch read and write operations.

Each configured Avalon-MM bidirectional port consists of the signals listed in the following table.

Table 11-15: Avalon-MM Bidirectional Port Signals

Name	Bit Width	Input/Output Direction	Function
clk	1	In	Clock for the Avalon-MM interface
read	1	In	Indicates read transaction ⁽²⁶⁾
write	1	In	Indicates write transaction ⁽²⁶⁾
address	32	In	Address of the transaction
readdata	32, 64, 128, or 256	Out	Read data return
readdatavalid	1	Out	Indicates the readdata signal contains valid data in response to a previous read request.
writedata	32, 64, 128, or 256	In	Write data for a transaction
byteenable	4, 8, 16, 32	In	Byte enables for each write byte lane

⁽²⁶⁾ The Avalon-MM protocol does not allow read and write transactions to be posted concurrently.

Name	Bit Width	Input/Output Direction	Function
waitrequest	1	Out	Indicates need for additional cycles to complete a transaction
burstcount	11	In	Transaction burst length. The value of the maximum burstcount parameter must be a power of 2.

The read and write interfaces are configured to the same size. The byte-enable size scales with the data bus size.

Related Information

[Avalon Interface Specifications](#)

Information about the Avalon-MM protocol

Avalon-MM Write-Only Port

The Avalon-MM write-only ports are standard Avalon-MM ports used to dispatch write operations. Each configured Avalon-MM write port consists of the signals listed in the following table.

Table 11-16: Avalon-MM Write-Only Port Signals

Name	Bits	Direction	Function
reset	1	In	Reset
clk	1	In	Clock
write	1	In	Indicates write transaction
address	32	In	Address of the transaction
writedata	32, 64, 128, or 256	In	Write data for a transaction
byteenable	4, 8, 16, 32	In	Byte enables for each write byte
waitrequest	1	Out	Indicates need for additional cycles to complete a transaction
burstcount	11	In	Transaction burst length

Related Information[Avalon Interface Specifications](#)

Information about the Avalon-MM protocol

Avalon-MM Read Port

The Avalon-MM read ports are standard Avalon-MM ports used only to dispatch read operations. Each configured Avalon-MM read port consists of the signals listed in the following table.

Table 11-17: Avalon-MM Read Port Signals

Name	Bits	Direction	Function
reset	1	In	Reset
clk	1	In	Clock
read	1	In	Indicates read transaction
address	32	In	Address of the transaction
readdata	32, 64, 128, or 256	Out	Read data return
readdatavalid	1	Out	Flags valid cycles for read data return
waitrequest	1	Out	Indicates the need for additional cycles to complete a transaction. Needed for read operations when delay is needed to accept the read command.
burstcount	11	In	Transaction burst length

Related Information[Avalon Interface Specifications](#)

Information about the Avalon-MM protocol

AXI Port

The AXI port uses an AXI-3 interface. Each configured AXI port consists of the signals listed in the following table. Because the AXI protocol allows simultaneous read and write commands to be issued, two SDRAM control ports are required to form an AXI interface.

Table 11-18: AXI Port Signals

Name	Bits	Direction	Channel	Function
ARESETn	1	In	n/a	Reset

Name	Bits	Direction	Channel	Function
ACLK	1	In	n/a	Clock
AWID	4	In	Write address	Write identification tag
AWADDR	32	In	Write address	Write address
AWLEN	4	In	Write address	Write burst length
AWSIZE	3	In	Write address	Width of the transfer size
AWBURST	2	In	Write address	Burst type
AWLOCK	2	In	Write address	Lock type signal which indicates if the access is exclusive; valid values are 0x0 (normal access) and 0x1 (exclusive access)
AWCACHE	4	In	Write address	Cache policy type
AWPROT	3	In	Write address	Protection-type signal used to indicate whether a transaction is secure or non-secure
AWREADY	1	Out	Write address	Indicates ready for a write command
AWVALID	1	In	Write address	Indicates valid write command.
WID	4	In	Write data	Write data transfer ID
WDATA	32, 64, 128 or 256	In	Write data	Write data
WSTRB	4, 8, 16, 32	In	Write data	Byte-based write data strobe. Each bit width corresponds to 8 bit wide transfer for 32-bit wide to 256-bit wide transfer.
WLAST	1	In	Write data	Last transfer in a burst
WVALID	1	In	Write data	Indicates write data and strobes are valid
WREADY	1	Out	Write data	Indicates ready for write data and strobes
BID	4	Out	Write response	Write response transfer ID
BRESP	2	Out	Write response	Write response status
BVALID	1	Out	Write response	Write response valid signal
BREADY	1	In	Write response	Write response ready signal
ARID	4	In	Read address	Read identification tag
ARADDR	32	In	Read address	Read address
ARLEN	4	In	Read address	Read burst length
ARSIZE	3	In	Read address	Width of the transfer size
ARBURST	2	In	Read address	Burst type
ARLOCK	2	In	Read address	Lock type signal which indicates if the access is exclusive; valid values are 0x0 (normal access) and 0x1 (exclusive access)

Name	Bits	Direction	Channel	Function
ARCACHE	4	In	Read address	Lock type signal which indicates if the access is exclusive; valid values are 0x0 (normal access) and 0x1 (exclusive access)
ARPROT	3	In	Read address	Protection-type signal used to indicate whether a transaction is secure or non-secure
ARREADY	1	Out	Read address	Indicates ready for a read command
ARVALID	1	In	Read address	Indicates valid read command
RID	4	Out	Read data	Read data transfer ID
RDATA	32, 64, 128 or 256	Out	Read data	Read data
RRESP	2	Out	Read data	Read response status
RLAST	1	Out	Read data	Last transfer in a burst
RVALID	1	Out	Read data	Indicates read data is valid
RREADY	1	In	Read data	Read data channel ready signal

Related Information**[Arm AMBA* Open Specification](#)**

AMBA Open Specifications, including information about the AXI-3 interface

SDRAM Controller Subsystem Programming Model

SDRAM controller configuration occurs through software programming of the configuration registers using the CSR interface.

HPS Memory Interface Architecture

The configuration and initialization of the memory interface by the Arm processor is a significant difference compared to the FPGA memory interfaces, and results in several key differences in the way the HPS memory interface is defined and configured.

Boot-up configuration of the HPS memory interface is handled by the initial software boot code, not by the FPGA programmer, as is the case for the FPGA memory interfaces. The Intel Quartus® Prime software is involved in defining the configuration of I/O ports which is used by the boot-up code, as well as timing analysis of the memory interface. Therefore, the memory interface must be configured with the correct PHY-level timing information. Although configuration of the memory interface in Platform Designer (Standard) is still necessary, it is limited to PHY- and board-level settings.

HPS Memory Interface Configuration

To configure the external memory interface components of the HPS, open the HPS interface by selecting the Arria V/Cyclone V Hard Processor System component in Platform Designer (Standard). Within the HPS interface, select the EMIF tab to open the EMIF parameter editor.

The EMIF parameter editor contains four additional tabs: PHY Settings, Memory Parameters, Memory Timing, and Board Settings. The parameters available on these tabs are similar to those available in the parameter editors for non-SoC device families.

There are significant differences between the EMIF parameter editor for the Hard Processor System and the parameter editors for non-SoC devices, as follows:

- Because the HPS memory controller is not configurable through the Intel Quartus Prime software, the Controller and Diagnostic tabs, which exist for non-SoC devices, are not present in the EMIF parameter editor for the hard processor system.
- Unlike the protocol-specific parameter editors for non-SoC devices, the EMIF parameter editor for the Hard Processor System supports multiple protocols, therefore there is an SDRAM Protocol parameter, where you can specify your external memory interface protocol. By default, the EMIF parameter editor assumes the DDR3 protocol, and other parameters are automatically populated with DDR3-appropriate values. If you select a protocol other than DDR3, change other associated parameter values appropriately.
- Unlike the memory interface clocks in the FPGA, the memory interface clocks for the HPS are initialized by the boot-up code using values provided by the configuration process. You may accept the values provided by UniPHY, or you may use your own PLL settings. If you choose to specify your own PLL settings, you must indicate that the clock frequency that UniPHY should use is the requested clock frequency, and not the achieved clock frequency calculated by UniPHY.

Note: The HPS does not support EMIF synthesis generation, compilation, or timing analysis. The HPS hard memory controller cannot be bonded with another hard memory controller on the FPGA portion of the device.

HPS Memory Interface Simulation

Platform Designer (Standard) provides a complete simulation model of the HPS memory interface controller and PHY, providing cycle-level accuracy, comparable to the simulation models for the FPGA memory interface.

The simulation model supports only the skip-cal simulation mode; quick-cal and full-cal are not supported. An example design is not provided. However, you can create a test design by adding the traffic generator component to your design using Platform Designer (Standard). Also, the HPS simulation model does not use external memory pins to connect to the DDR memory model; instead, the memory model is incorporated directly into the HPS SDRAM interface simulation modules. The memory instance incorporated into the HPS model is in the simulation model hierarchy at: `hps_0/fpga_interfaces/f2sdram/hps_sdram_inst/mem/`

Simulation of the FPGA-to-SDRAM interfaces requires that you first bring the interfaces out of reset, otherwise transactions cannot occur. Connect the H2F reset to the F2S port resets and add a stage to your testbench to assert and deassert the H2F reset in the HPS. Appropriate Verilog code is shown below:

```
initial
begin
    // Assert reset
    <base name>.hps.fpga_interfaces.h2f_reset_inst.reset_assert();
    // Delay
    #1
    // Deassert reset
    <base name>.hps.fpga_interfaces.h2f_reset_inst.reset_deassert();
end
```

Generating a Preloader Image for HPS with EMIF

To generate a preloader image for an HPS-based external memory interface, you must complete the following tasks:

- Create a project in Platform Designer (Standard).
- Create a top-level file and add constraints.
- Create a preloader BSP file.
- Create a preloader image.

Creating a Project in Platform Designer (Standard)

Before you can generate a preloader image, you must create a project in Platform Designer (Standard), as follows:

1. On the **Tools** menu in the Intel Quartus Prime software, click **Platform Designer (Standard)**.
2. Under **Component library**, expand **Embedded Processor System**, select **Hard Processor System** and click **Add**.
3. Specify parameters for the **FPGA Interfaces**, **Peripheral Pin Multiplexing**, and **HPS Clocks**, based on your design requirements.
4. On the **SDRAM** tab, select the SDRAM protocol for your interface.
5. Populate the necessary parameter fields on the **PHY Settings**, **Memory Parameters**, **Memory Timing**, and **Board Settings** tabs.
6. Add other components in your design and make the appropriate bus connections.
7. Save the project.
8. Click **Generate** on the **Generation** tab. Platform Designer (Standard) generates the design.

Creating a Top-Level File and Adding Constraints

This topic describes adding your Platform Designer (Standard) system to your top-level design and adding constraints to your design.

1. Add your Platform Designer (Standard) system to your top-level design.
2. Add the Intel Quartus Prime IP files (.qip) generated in step 2, to your Intel Quartus Prime project.
3. Perform analysis and synthesis on your design.
4. Constrain your EMIF design by running the `<variation_name>_p0_pin_assignments.tcl` pin constraints script.
5. Add other necessary constraints—such as timing constraints, location assignments, and pin I/O standard assignments—for your design.
6. Compile your design to generate an SRAM object file (.sof) and the hardware handoff files necessary for creating a preloader image.

Note: You must regenerate the hardware handoff files whenever the HPS configuration changes; for example, due to changes in Peripheral Pin Multiplexing or I/O standard for HPS pins.

Related Information

[Altera SoC Embedded Design Suite User Guide](#)

For more information on how to create a preloader BSP file and image.

Debugging HPS SDRAM in the Preloader

To assist in debugging your design, tools are available at the preloader stage.

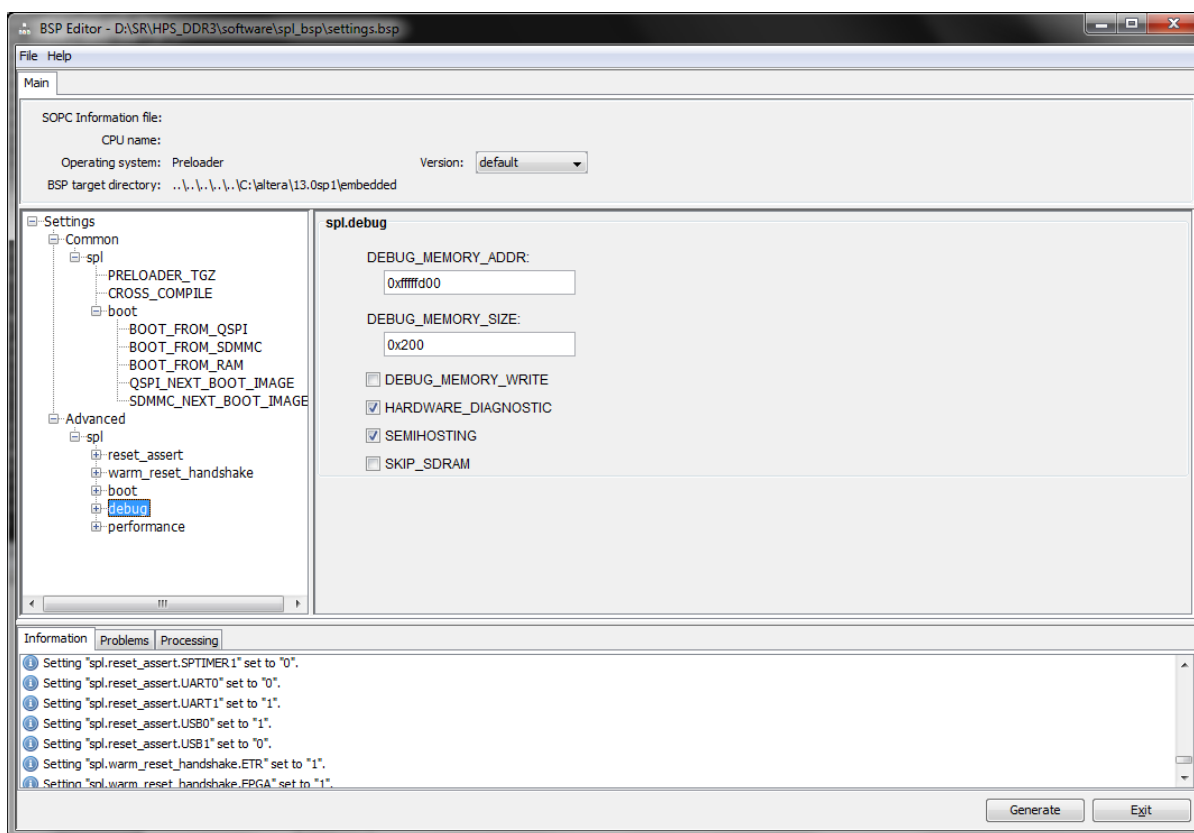
- UART or semihosting printout
- Simple memory test
- Debug report
- Predefined data patterns

The following topics provide procedures for implementing each of the above tools.

Enabling UART or Semihosting Printout

UART printout is enabled by default. If UART is not available on your system, you can use semihosting together with the debugger tool. To enable semihosting in the Preloader, follow these steps:

1. When you create the .bsp file in the BSP Editor, select **SEMIHOSTING** in the **spl.debug** window.



2. Enable semihosting in the debugger, by typing `set semihosting enabled true` at the command line in the debugger.


```

Commands History Scripts
Linked: New_configuration (1)

No SYSID registers could be found. Has a peripheral description file been supplied?

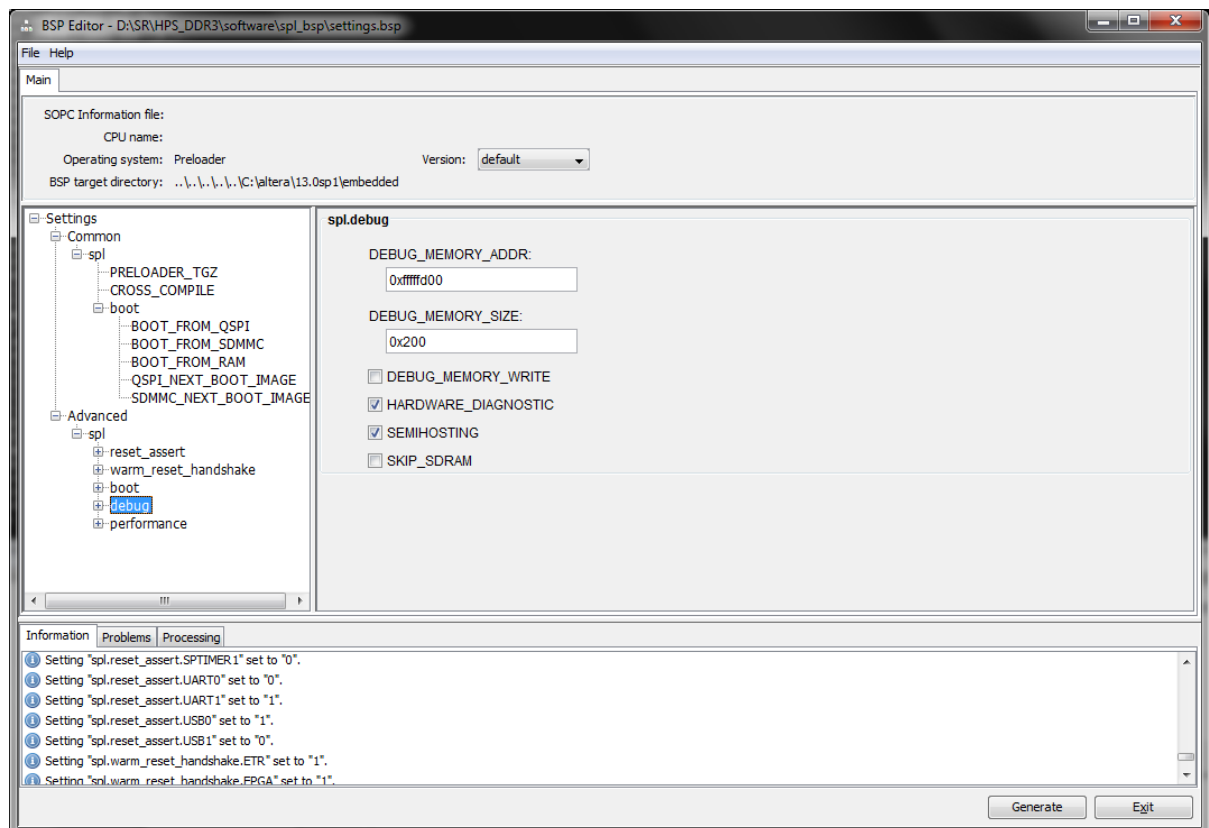
cd "C:\Users\kigtan\Documents\DS-5 Workspace_GHRD"
Working directory "C:\Users\kigtan\Documents\DS-5 Workspace_GHRD"
reset reset.system
Target has been reset
Execution stopped due to a breakpoint or watchpoint: S:0x00000000
S:0x00000000 LDR pc,[pc,#24] ; [0x20] = 0xA8
loadfile "D:\SR\HPS_DDR3\software\spl_bsp\uboot-socfpga\spl\uboot-spl"
Loaded section .text: S:0xFFFF0000 ~ S:0xFFFF8807 (size 0x8808)
Loaded section .rodata: S:0xFFFF8808 ~ S:0xFFFFA501 (size 0x1CFA)
Loaded section .data: S:0xFFFFA508 ~ S:0xFFFFB4EF (size 0xFE8)
Loaded section .bss: S:0xFFFFB4F0 ~ S:0xFFFFB593 (size 0xA4)
Loaded section .malloc: S:0xFFFFB594 ~ S:0xFFFFC993 (size 0x1400)
Loaded section .stack: S:0xFFFFC994 ~ S:0xFFFFD997 (size 0x1004)
Loaded section .spl_irq_stack: S:0xFFFFD998 ~ S:0xFFFFE99F (size 0x1008)
Entry point S:0xFFFF0000
set semihosting enabled true
Semihosting server socket created at port 8000
Command: set semihosting enabled true
Submit

```

Enabling Simple Memory Test

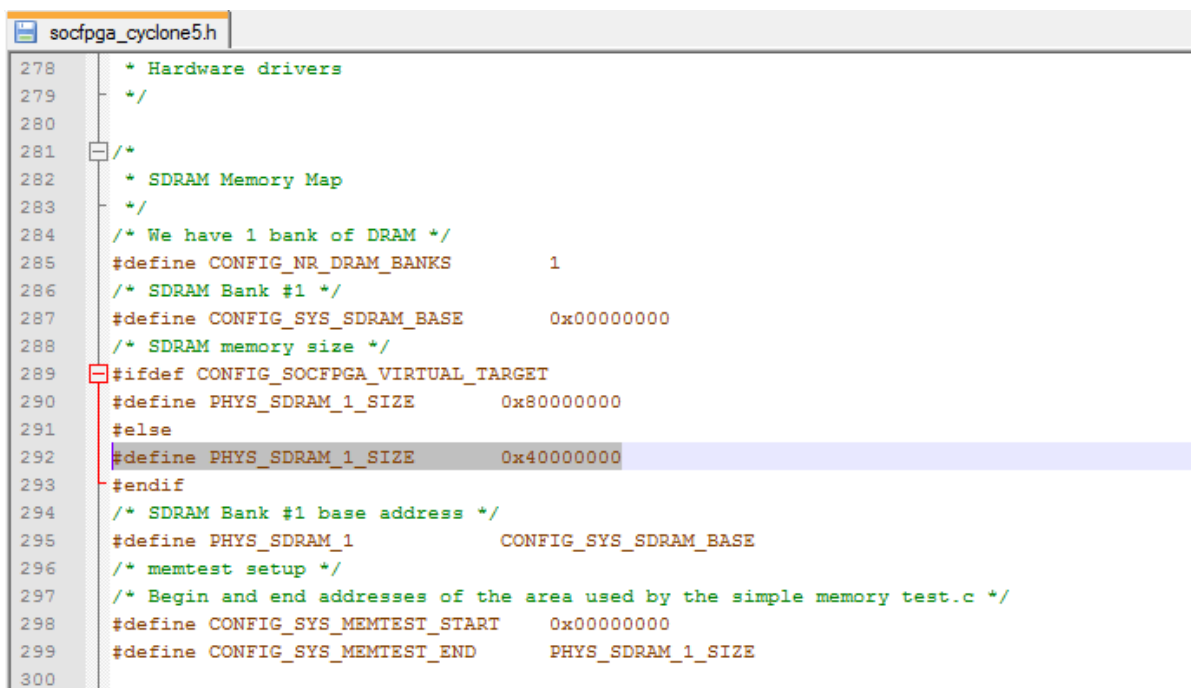
After the SDRAM is successfully calibrated, a simple memory test may be performed using the debugger.

1. When you create the .bsp file in the BSP Editor, select **HARDWARE_DIAGNOSTIC** in the **spl.debug** window..



2. The simple memory test assumes SDRAM with a memory size of 1 GB. If your board contains a different SDRAM memory size, open the file <design folder>\spl_bsp\uboot-socfpga\

include\configs\socfpga_arria5.h in a text editor, and change the PHYS_SDRAM_1_SIZE parameter at line 292 to specify your actual memory size in bytes.



```
278  * Hardware drivers
279  */
280
281  /*
282  * SDRAM Memory Map
283  */
284  /* We have 1 bank of DRAM */
285  #define CONFIG_NR_DRAM_BANKS      1
286  /* SDRAM Bank #1 */
287  #define CONFIG_SYS_SDRAM_BASE      0x00000000
288  /* SDRAM memory size */
289  #ifndef CONFIG_SOCFPGA_VIRTUAL_TARGET
290  #define PHYS_SDRAM_1_SIZE          0x80000000
291  #else
292  #define PHYS_SDRAM_1_SIZE          0x40000000
293  #endif
294  /* SDRAM Bank #1 base address */
295  #define PHYS_SDRAM_1              CONFIG_SYS_SDRAM_BASE
296  /* memtest setup */
297  /* Begin and end addresses of the area used by the simple memory test.c */
298  #define CONFIG_SYS_MEMTEST_START    0x00000000
299  #define CONFIG_SYS_MEMTEST_END      PHYS_SDRAM_1_SIZE
300
```

Enabling the Debug Report

You can enable the SDRAM calibration sequencer to produce a debug report on the UART printout or semihosting output. To enable the debug report, follow these steps:

1. After you have enabled the UART or semihosting, open the file `<project_directory>\hps_isw_handoff\sequencer_defines.hin` in a text editor.
2. Locate the line `#define RUNTIME_CAL_REPORT 0` and change it to `#define RUNTIME_CAL_REPORT 1`.

Figure 11-9: Semihosting Printout With Debug Support Enabled

```

DS-5 Debug - Eclipse Platform
File Edit Navigate Search Project Run Window Help
App Console Error Log
U-Boot SPL 2012.10 (Sep 09 2013 - 19:25:45)
SDRAM: Initializing MRR registers
SDRAM: calibrating PHY
SEQ.C: Preparing to start memory calibration
SEQ.C: DQS Enable ; Group 0 ; Rank 0 ; Start VFIFO 5 ; Phase 5 ; Delay 3
SEQ.C: DQS Enable ; Group 0 ; Rank 0 ; End VFIFO 6 ; Phase 4 ; Delay 14
SEQ.C: DQS Enable ; Group 0 ; Rank 0 ; Center VFIFO 6 ; Phase 1 ; Delay 3
SEQ.C: Read Deskew ; DQ 0 ; Rank 0 ; Left edge 17 ; Right edge 27 ; DQ delay 3 ; DQS delay 12
SEQ.C: Read Deskew ; DQ 1 ; Rank 0 ; Left edge 30 ; Right edge 23 ; DQ delay 11 ; DQS delay 12
SEQ.C: Read Deskew ; DQ 2 ; Rank 0 ; Left edge 13 ; Right edge 27 ; DQ delay 1 ; DQS delay 12
SEQ.C: Read Deskew ; DQ 3 ; Rank 0 ; Left edge 30 ; Right edge 24 ; DQ delay 11 ; DQS delay 12
SEQ.C: Read Deskew ; DQ 4 ; Rank 0 ; Left edge 17 ; Right edge 27 ; DQ delay 3 ; DQS delay 12
SEQ.C: Read Deskew ; DQ 5 ; Rank 0 ; Left edge 31 ; Right edge 24 ; DQ delay 11 ; DQS delay 12
SEQ.C: Read Deskew ; DQ 6 ; Rank 0 ; Left edge 11 ; Right edge 27 ; DQ delay 0 ; DQS delay 12
SEQ.C: Read Deskew ; DQ 7 ; Rank 0 ; Left edge 30 ; Right edge 23 ; DQ delay 11 ; DQS delay 12
SEQ.C: Write Deskew ; DQ 0 ; Rank 0 ; Left edge 31 ; Right edge 13 ; DQ delay 9 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 1 ; Rank 0 ; Left edge 31 ; Right edge 14 ; DQ delay 9 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 2 ; Rank 0 ; Left edge 29 ; Right edge 19 ; DQ delay 5 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 3 ; Rank 0 ; Left edge 29 ; Right edge 17 ; DQ delay 6 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 4 ; Rank 0 ; Left edge 31 ; Right edge 14 ; DQ delay 9 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 5 ; Rank 0 ; Left edge 31 ; Right edge 14 ; DQ delay 9 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 6 ; Rank 0 ; Left edge 28 ; Right edge 19 ; DQ delay 5 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 7 ; Rank 0 ; Left edge 29 ; Right edge 17 ; DQ delay 6 ; DQS delay 4
SEQ.C: DM Deskew ; Group 0 ; Left edge 27 ; Right edge 18 ; DM delay 4
SEQ.C: Read after Write ; DQ 0 ; Rank 0 ; Left edge 24 ; Right edge 19 ; DQ delay 3 ; DQS delay 12
SEQ.C: Read after Write ; DQ 1 ; Rank 0 ; Left edge 31 ; Right edge 9 ; DQ delay 11 ; DQS delay 12
SEQ.C: Read after Write ; DQ 2 ; Rank 0 ; Left edge 19 ; Right edge 19 ; DQ delay 0 ; DQS delay 12
SEQ.C: Read after Write ; DQ 3 ; Rank 0 ; Left edge 31 ; Right edge 9 ; DQ delay 11 ; DQS delay 12
SEQ.C: Read after Write ; DQ 4 ; Rank 0 ; Left edge 24 ; Right edge 19 ; DQ delay 3 ; DQS delay 12
SEQ.C: Read after Write ; DQ 5 ; Rank 0 ; Left edge 31 ; Right edge 8 ; DQ delay 12 ; DQS delay 12
SEQ.C: Read after Write ; DQ 6 ; Rank 0 ; Left edge 18 ; Right edge 19 ; DQ delay 0 ; DQS delay 12
SEQ.C: Read after Write ; DQ 7 ; Rank 0 ; Left edge 31 ; Right edge 8 ; DQ delay 12 ; DQS delay 12
SEQ.C: DQS Enable ; Group 1 ; Rank 0 ; Start VFIFO 5 ; Phase 5 ; Delay 13
SEQ.C: DQS Enable ; Group 1 ; Rank 0 ; End VFIFO 6 ; Phase 4 ; Delay 9
SEQ.C: DQS Enable ; Group 1 ; Rank 0 ; Center VFIFO 6 ; Phase 0 ; Delay 11
SEQ.C: Read Deskew ; DQ 8 ; Rank 0 ; Left edge 16 ; Right edge 27 ; DQ delay 1 ; DQS delay 11
SEQ.C: Read Deskew ; DQ 9 ; Rank 0 ; Left edge 29 ; Right edge 26 ; DQ delay 8 ; DQS delay 11
SEQ.C: Read Deskew ; DQ 10 ; Rank 0 ; Left edge 13 ; Right edge 27 ; DQ delay 0 ; DQS delay 11
SEQ.C: Read Deskew ; DQ 11 ; Rank 0 ; Left edge 30 ; Right edge 24 ; DQ delay 10 ; DQS delay 11
SEQ.C: Read Deskew ; DQ 12 ; Rank 0 ; Left edge 16 ; Right edge 27 ; DQ delay 1 ; DQS delay 11
SEQ.C: Read Deskew ; DQ 13 ; Rank 0 ; Left edge 30 ; Right edge 24 ; DQ delay 10 ; DQS delay 11
SEQ.C: Read Deskew ; DQ 14 ; Rank 0 ; Left edge 13 ; Right edge 27 ; DQ delay 0 ; DQS delay 11
SEQ.C: Read Deskew ; DQ 15 ; Rank 0 ; Left edge 30 ; Right edge 25 ; DQ delay 9 ; DQS delay 11
SEQ.C: Write Deskew ; DQ 8 ; Rank 0 ; Left edge 31 ; Right edge 15 ; DQ delay 8 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 9 ; Rank 0 ; Left edge 31 ; Right edge 16 ; DQ delay 7 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 10 ; Rank 0 ; Left edge 29 ; Right edge 16 ; DQ delay 6 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 11 ; Rank 0 ; Left edge 29 ; Right edge 16 ; DQ delay 6 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 12 ; Rank 0 ; Left edge 31 ; Right edge 14 ; DQ delay 8 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 13 ; Rank 0 ; Left edge 31 ; Right edge 16 ; DQ delay 7 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 14 ; Rank 0 ; Left edge 27 ; Right edge 17 ; DQ delay 5 ; DQS delay 4
SEQ.C: Write Deskew ; DQ 15 ; Rank 0 ; Left edge 29 ; Right edge 17 ; DQ delay 6 ; DQS delay 4
SEQ.C: DM Deskew ; Group 1 ; Left edge 27 ; Right edge 18 ; DM delay 4
SEQ.C: Read after Write ; DQ 8 ; Rank 0 ; Left edge 21 ; Right edge 20 ; DQ delay 1 ; DQS delay 12
SEQ.C: Read after Write ; DQ 9 ; Rank 0 ; Left edge 31 ; Right edge 11 ; DQ delay 11 ; DQS delay 12
SEQ.C: Read after Write ; DQ 10 ; Rank 0 ; Left edge 19 ; Right edge 20 ; DQ delay 0 ; DQS delay 12
SEQ.C: Read after Write ; DQ 11 ; Rank 0 ; Left edge 31 ; Right edge 9 ; DQ delay 12 ; DQS delay 12
SEQ.C: Read after Write ; DQ 12 ; Rank 0 ; Left edge 21 ; Right edge 20 ; DQ delay 1 ; DQS delay 12

```

Analysis of Debug Report

The following analysis helps you interpret the debug report.

- The Read Deskew and Write Deskew results shown in the debug report are before calibration. (Before calibration results are actually from the window seen *during* calibration, and are most useful for debugging.)
- For each DQ group, the Write Deskew, Read Deskew, DM Deskew, and Read after Write results map to the before-calibration margins reported in the EMIF Debug Toolkit.

Note: The Write Deskew, Read Deskew, DM Deskew, and Read after Write results are reported in delay steps (nominally 25ps, in Arria V and Cyclone V devices), not in picoseconds.

- DQS Enable calibration is reported as a VFIFO setting (in one clock period steps), a phase tap (in one-eighth clock period steps), and a delay chain step (in 25ps steps).

```
SEQ.C: DQS Enable ; Group 0 ; Rank 0 ; Start VFIFO 5 ; Phase 6 ; Delay 4
SEQ.C: DQS Enable ; Group 0 ; Rank 0 ; End VFIFO 6 ; Phase 5 ; Delay 9
SEQ.C: DQS Enable ; Group 0 ; Rank 0 ; Center VFIFO 6 ; Phase 2 ; Delay 1
```

Analysis of DQS Enable results: A VFIFO tap is 1 clock period, a phase is 1/8 clock period (45 degrees) and delay is nominally 25ps per tap. The DQSen window is the difference between the start and end—for the above example, assuming a frequency of 400 MHz (2500ps), that calculates as follows: start is $5 * 2500 + 6 * 2500 / 8 + 4 * 25 = 14475\text{ps}$. By the same calculation, the end is 16788ps. Consequently, the DQSen window is 2313ps.

- The size of a read window or write window is equal to (left edge + right edge) * delay chain step size. Both the left edge and the right edge can be negative or positive.:

```
SEQ.C: Read Deskew ; DQ 0 ; Rank 0 ; Left edge 18 ; Right edge 27 ; DQ
delay 0 ; DQS delay 8
SEQ.C: Write Deskew ; DQ 0 ; Rank 0 ; Left edge 30 ; Right edge 17 ; DQ
delay 6 ; DQS delay 4
```

Analysis of DQ and DQS delay results: The DQ and DQS output delay (write) is the D5 delay chain. The DQ input delay (read) is the D1 delay chain, the DQS input delay (read) is the D4 delay chain.

- Consider the following example of latency results:

```
SEQ.C: LFIFO Calibration ; Latency 10
```

Analysis of latency results: This is the calibrated PHY read latency. The EMIF Debug Toolkit does not report this figure. This latency is reported in clock cycles.

- Consider the following example of FOM results:

```
SEQ.C: FOM IN = 83
SEQ.C: FOM OUT = 91
```

Analysis of FOM results: The FOM IN value is a measure of the health of the read interface; it is calculated as the sum over all groups of the minimum margin on DQ plus the margin on DQS, divided by 2. The FOM OUT is a measure of the health of the write interface; it is calculated as the sum over all groups of the minimum margin on DQ plus the margin on DQS, divided by 2. You may refer to these values as indicators of improvement when you are experimenting with various termination schemes, assuming there are no individual misbehaving DQ pins.

- The debug report does not provide delay chain step size values. The delay chain step size varies with device speed grade. Refer to your device data sheet for exact incremental delay values for delay chains.

Related Information

Functional Description—UniPHY

For more information about calibration, refer to the *Calibration Stages* section in the *Functional Description—UniPHY* chapter of the *External Memory Interface Handbook Volume 3: Reference Material*.

Writing a Predefined Data Pattern to SDRAM in the Preloader

You can include your own code to write a predefined data pattern to the SDRAM in the preloader for debugging purposes.

1. Include your code in the file: `<project_folder>\software\spl_bsp\uboot-socfpga\arch\arm\cpu\armv7\socfpga\spl.c`.

Adding the following code to the `spl.c` file causes the controller to write walking 1s and walking 0s, repeated five times, to the SDRAM.

```
/*added for demo, place after the last #define statement in spl.c */
#define ROTATE_RIGHT(X) ( (X>>1) | (X&1?0X80000000:0) )
/*added for demo, place after the calibration code */
test_data_walk0((long *)0x100000,PHYS_SDRAM_1_SIZE);
int test_data_walk0(long *base, long maxsize)
{
    volatile long *addr;
    long cnt;
    ulong data_temp[3];
    ulong expected_data[3];
    ulong read_data;
    int i = 0; //counter to loop different data pattern
    int num_address;

    num_address=50;

    data_temp[0]=0xFFFFFFFF; //initial data for walking 0 pattern
    data_temp[1]=0X00000001; //initial data for walking 1 pattern
    data_temp[2]=0XAAAAAAAA; //initial data for A->5 switching

    expected_data[0]=0xFFFFFFFF; //initial data for walking 0 pattern
    expected_data[1]=0X00000001; //initial data for walking 1 pattern
    expected_data[2]=0XAAAAAAAA; //initial data for A->5 switching

    for (i=0;i<3;i++) {
        printf("\nSTARTED %08X DATA PATTERN !!!!\n",data_temp[i]);
        /*write*/
        for (cnt = (0+i*num_address); cnt < ((i+1)*num_address) ; cnt++ ) {
            addr = base + cnt; /* pointer arith! */
            sync ();
            *addr = data_temp[i];
            data_temp[i]=ROTATE_RIGHT(data_temp[i]);
        }

        /*read*/
        for (cnt = (0+i*num_address); cnt < ((i+1)*num_address) ; cnt = cnt++ ) {
            addr = base + cnt; /* pointer arith! */
            sync ();
            read_data=*addr;
            printf("Address:%X Expected: %08X Read:%08X \n",addr,
            expected_data[i],read_data);
            if (expected_data[i] !=read_data) {
                puts("!!!!!!FAILED!!!!!!\n\n");
                hang();
            }
            expected_data[i]=ROTATE_RIGHT(expected_data[i]);
        }
    }
}
}
}
}
}
====//End Of Code//=====
```

[illegible]

Base addresses of all HPS modules

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) contains two types of on-chip memory. The on-chip memory types are:

- On-chip RAM—The on-chip RAM provides 64 KB of general-purpose RAM
- Boot ROM—The boot ROM contains the code required to boot the HPS from cold or warm reset

Both on-chip memories connect to the level 3 (L3) interconnect.

The hard processor system (HPS) contains on-chip RAM. The on-chip RAM provides 64 KB of general-purpose single-port RAM and connects to the level 3 (L3) interconnect.

Related Information

- [On-Chip RAM](#) on page 12-1
- [Boot ROM](#) on page 12-3

On-Chip RAM

Features of the On-Chip RAM

The on-chip RAM offers the following features:

- 64-bit slave interface
- 64 KB of single-ported RAM
- Read acceptance of two, write acceptance of two, and a total acceptance of four
- Error correction code (ECC) support
- High throughput - read or write every clock cycle.

Related Information

[Clock Manager](#) on page 3-1

On-Chip RAM Block Diagram and System Integration

Transfers between memory and the NIC-301 L3 interconnect happen through a 64-bit interface, gated by the `l3_main_clk` interconnect clock. ECC logic detects single-bit, corrected and double-bit, uncorrected errors. For memory, read acceptance is two, write acceptance is two, and total acceptance is two with a round-robin arbitration.

The entire RAM is either secure or non-secure. Security is enforced by the NIC-301 L3 interconnect.

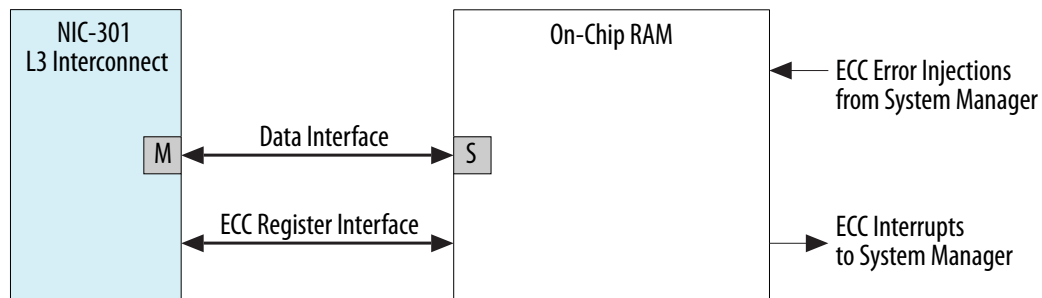
Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

Figure 12-1: On-Chip RAM Block Diagram



Note: You must initialize the on-chip RAM before you enable the ECC support to prevent false ECC interrupts triggered by uninitialized bits.

Functional Description of the On-Chip RAM

The on-chip RAM uses an 64-bit slave interface. The slave interface supports transfers between memory and the NIC-301 L3 interconnect. All reads and writes are serviced in order.

Related Information

[Clock Manager](#) on page 3-1

On-Chip RAM Clocks

The on-chip RAM is driven by the `l3_main_clk` interconnect clock.

The on-chip RAM uses an 64-bit slave interface. The slave interface supports transfers between memory and the NIC-301 L3 interconnect. All reads and writes are serviced in order.

On-Chip RAM Resets

On-Chip RAM Initialization

You must initialize the on-chip RAM before you enable the ECC. Failure to do so triggers spurious interrupts.

Follow these steps to enable ECC:

1. Turn on the ECC hardware, but disable the interrupts.
2. Initialize the SRAM in the NAND.
3. Clear the ECC event bits, because these bits may have become asserted after Step 1.
4. Enable the ECC interrupts now that the ECC bits have been set.

Initialize the on-chip RAM using the following steps:

- Disable ECC interrupts
- Enable ECC generation
- Initialize memory by clearing the contents by writing 0x0 in the address space
- Enable ECC interrupts

Related Information

- [On-Chip Memory Address Map and Register Definitions](#) on page 12-4
 - [System Manager](#) on page 6-1
- For more information about ECC, refer to the *System Manager* chapter of the Arria V Device Handbook.

Boot ROM

Features of the Boot ROM

The boot ROM offers the following features:

- 32-bit interface
- 64 KB size
- Single-ported ROM
- Read acceptance of two
- High throughput - read every clock cycle.

Related Information

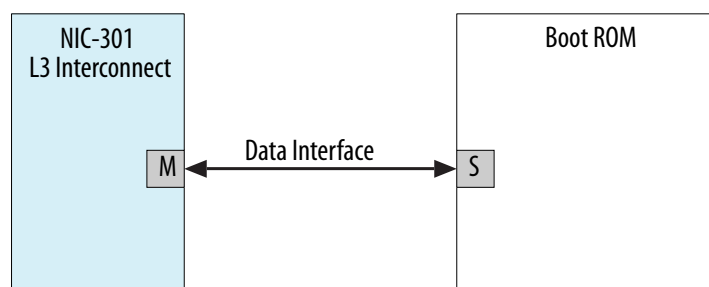
[Clock Manager](#) on page 3-1

Boot ROM Block Diagram and System Integration

Transfers between memory and the NIC-301 L3 interconnect happen through a 32-bit data interface, gated by the `l3_main_clk` interconnect clock.

The entire RAM is either secure or nonsecure. Security is enforced by the NIC-301 L3 interconnect.

Figure 12-2: Boot ROM Block Diagram



Functional Description of the Boot ROM

The boot ROM uses an 32-bit slave interface. The slave interface supports transfers between memory and the NIC-301 L3 interconnect. All writes return an error response.

Related Information

- [Clock Manager](#) on page 3-1
- [Booting and Configuration](#) on page 29-1

Boot ROM Clocks

Related Information

[Clock Manager](#) on page 3-1

Boot ROM Resets

The boot ROM reset is driven by the boot_rom_rst_n interconnect clock.

On-Chip Memory Address Map and Register Definitions

The address map and register definitions for the on-chip memories consist of the following regions:

- Boot ROM Module
- On-chip RAM Module

Related Information

[Introduction to the Hard Processor System](#) on page 2-1



2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) provides a NAND flash controller to interface with external NAND flash memory in Intel system-on-a-chip (SoC) systems. You can use external flash memory to store software, or as extra storage capacity for large applications or user data. The HPS NAND flash controller is based on the CadenceDesign IP* NAND Flash Memory Controller.

NAND Flash Controller Features

The NAND flash controller provides the following functionality and features:

- Supports one x8 NAND flash device
- Supports Open NAND Flash Interface (ONFI) 1.0
- Supports NAND flash memories from Hynix, Samsung, Toshiba, Micron, STMicroelectronics, and Spansion

Note: The Spansion S34ML08G2 NAND flash memory is verified to work properly with the HPS.

- Supports error correction codes (ECCs) providing single-bit error correction and double-bit error detection, with:
 - Sector size programmable 512 byte (4-, 8-, or 16-bit correction) or 1024 byte (24-bit correction)
 - Three NAND FIFOs - ECC Buffer, write FIFO and read FIFO
- Supports pipeline read-ahead and write commands for enhanced read and write throughput
- Supports devices with 32, 64, 128, 256, 384, or 512 pages per block
- Supports multiplane devices
- Supports page sizes of 512 bytes, 2 kilobytes (KB), 4 KB, or 8 KB
- Supports single layer cell (SLC) and multiple layer cell (MLC) devices with programmable correction capabilities
- Provides internal direct memory access (DMA)
- Provides programmable access timing

Related Information

Supported Flash Devices for Cyclone V and Arria V SoC

For more information, refer to the supported NAND flash devices section on this page.

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

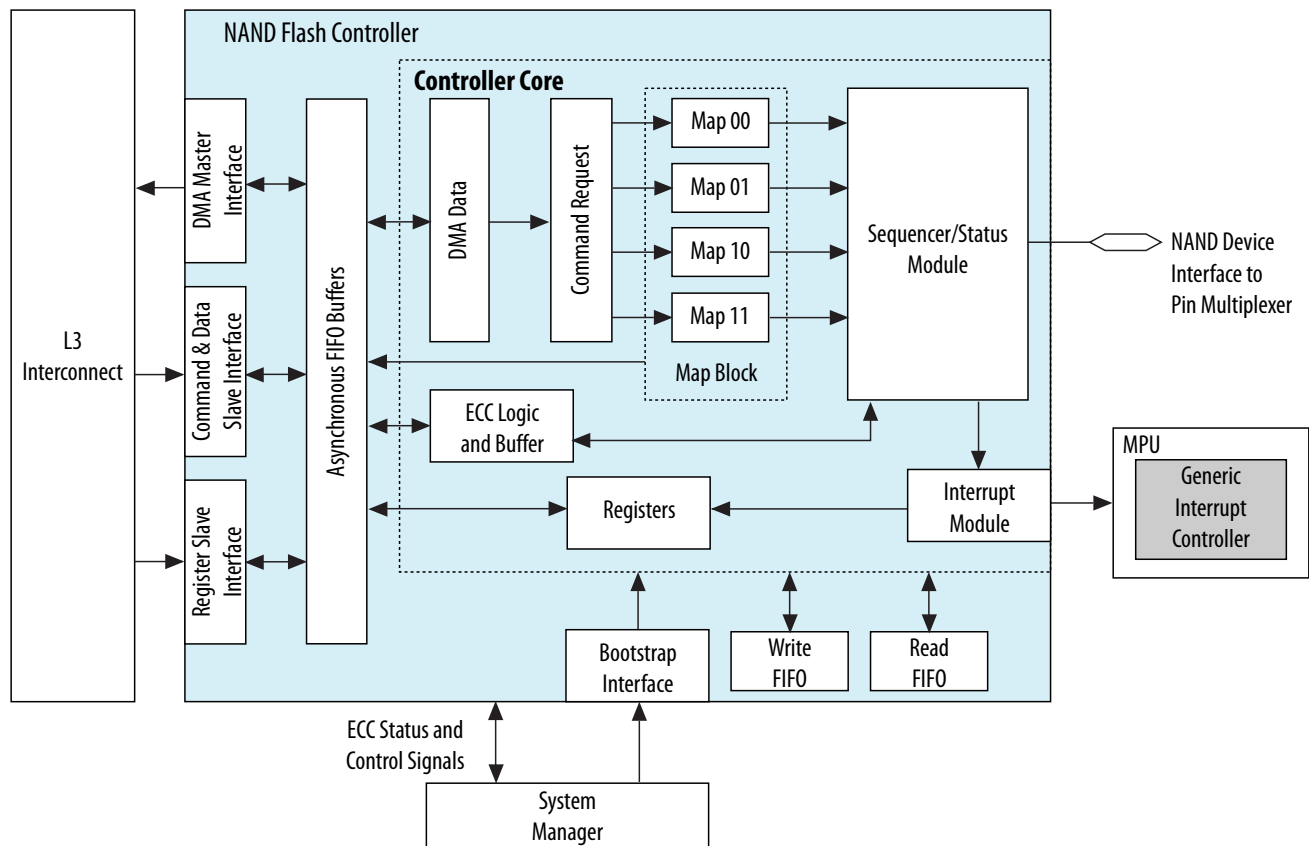
ISO
9001:2015
Registered

ALTERA
now part of Intel

NAND Flash Controller Block Diagram and System Integration

Figure 13-1: NAND Flash Controller Block Diagram

The following figure shows integration of the NAND flash controller in the HPS.



Features of the flash controller:

- Receives commands and data from the host through memory-mapped control and data registers connected to the command and data slave interface
- The host accesses the flash controller's control and status registers (CSRs) through the register slave interface.
- Handles all command sequencing and flash device interactions
- Generates interrupts to the HPS Cortex-A9 MPCore
- The DMA master interface provides accesses to and from the flash controller through the controller's built-in DMA.

NAND Flash Controller Signal Descriptions

The HPS I/O pins support a single x8 device. The following table lists the signals:

Table 13-1: NAND Flash Interface Signals

Signal	Width	I/O	Description
ad	8	in/out	Command, address and data for the flash device
ale	1	out	Address latch enable
ce_n	1	out	Output Active-low chip enable
cle	1	out	Command latch enable
re_n	1	out	Active-low read enable signal
rb	1	in	Ready/busy signal
we_n	1	out	Active-low write enable signal
wp_n	1	out	Active-low write protect signal

Functional Description of the NAND Flash Controller

This section describes the functionality of the NAND flash controller.

Discovery and Initialization

The NAND flash controller performs a specific initialization sequence after the HPS receives power and the flash device is stable. During initialization, the flash controller queries the flash device and configures itself according to one of the following flash device types:

- ONFI 1.0-compliant devices
- Legacy (non-ONFI) NAND devices

The NAND flash controller identifies ONFI-compliant connected devices using ONFI discovery protocol, by sending the Read ID command. For devices that do not recognize this command (especially for 512-byte page size devices), software must write to the system manager to assert the `bootstrap_512B_device` signal to identify the device type before releasing the NAND controller from reset.

To support booting and initialization, the `rdy_busy_in` pin must be connected.

The NAND flash controller performs the following initialization steps:

1. If the system manager is asserting `bootstrap_inhibit_init`, the flash controller goes directly to [step 7](#).
2. When the device is ready, the flash controller sends the "Read ID" command to read the ONFI signature from the memory device, to determine whether an ONFI or a legacy device is connected.
3. If the data returned by the memory device has an ONFI signature, the flash controller then reads the device parameter page. The flash controller stores the relevant device feature information in internal



memory control registers, enabling it to correctly program other registers in the flash device, and goes to [step 5](#).

4. If the data does not have a valid ONFI signature, the flash controller assumes that it is a legacy (non-ONFI) device. The flash controller then performs the following steps:
 - a. Sends the reset command to the device
 - b. Reads the device signature information
 - c. Stores the relevant values into internal memory controller registers
 5. The flash controller resets the memory device. At the same time, it verifies the width of the memory interface. The HPS supports one 8-bit NAND flash device. As a result, the flash controller always detects an 8-bit memory interface.
 6. The flash controller sends the Page Load command to block 0, page 0 of the device, configuring direct read access, so the processor can boot from that page. The processor can start reading from the first page of the flash memory, which is the expected location of the pre-loader software.
- Note:** The system manager can bypass this step by asserting `bootstrap_inhibit_b0p0_load` before reset is de-asserted.
7. The flash controller sends the reset command to the flash.
 8. The flash controller clears the `rst_comp` bit in the `intr_status0` register in the status group to indicate to software that the flash reset is complete.

Bootstrap Interface

The NAND flash controller provides a bootstrap interface that allows software to override the default behavior of the flash controller. The bootstrap interface contains four bits, which when set appropriately, allows the flash controller to skip the initialization phase and begin loading from flash memory immediately after reset. These bits are driven by software through the system manager. They are sampled by the NAND flash controller when the controller is released from reset.

Related Information

[System Manager](#) on page 6-1

For more information about the bootstrap interface control bits.

Bootstrap Setting Bits

The following table lists the relevant bootstrap setting bits, found in the system manager's bootstrap register, in the NAND flash controller register group. As an example, this table also lists recommended bootstrap settings for a 512-byte page device.

Table 13-2: Bootstrap Setting Bits

Bit	Example Value for 512-Byte Page
<code>noinit</code>	1 ⁽²⁷⁾
<code>page512</code>	1
<code>noloadb0p0</code>	1

⁽²⁷⁾ When this register is set, the NAND flash controller expects the host to program the related device parameter registers. For more information, refer to "Configuration by Host".

Bit	Example Value for 512-Byte Page
tworowaddr	1—flash device supports two-cycle addressing 0—flash device support three-cycle addressing

Related Information

[Configuration by Host](#) on page 13-5

Configuration by Host

If the system manager sets `bootstrap_inhibit_init` to 1, the NAND flash controller does not perform the process described in "Discovery and Initialization". In this case, the host processor must configure the flash controller.

When performance is not a concern in the design, the timing registers can be left unprogrammed.

Related Information

- [Bootstrap Setting Bits](#) on page 13-4
For recommended configuration-by-host settings to enable the basic read, write, and erase operations for a single-plane, 512 bytes/page device.
- [Discovery and Initialization](#) on page 13-3

Recommended Bootstrap Settings for 512-Byte Page Device

Table 13-3: Recommended Bootstrap Settings for an 8-bit, 512-Byte Page Device

Register ⁽²⁸⁾	Value
<code>devices_connected</code>	1
<code>device_width</code>	0 indicating an 8-bit NAND flash device
<code>number_of_planes</code>	1 indicating a single-plane device
<code>device_main_area_size</code>	The value of this register must reflect the flash device's page main area size.
<code>device_spare_area_size</code>	The value of this register must reflect the flash device's page spare area size.
<code>pages_per_block</code>	The value of this register must reflect number of pages per block in the flash device.

NAND Page Main and Spare Areas

Each NAND page has a main area and a spare area. The main area is intended for data storage. The spare area is intended for ECC and maintenance data, such as wear leveling information. Each block consists of a group of pages.

⁽²⁸⁾ All registers are in the config group.

The sizes of the main and spare areas, and the number of blocks in a page, depend on the specific NAND device connected to the NAND flash controller. Therefore, the device-dependent registers, `device_main_area_size`, `device_spare_area_size`, and `pages_per_block`, must be programmed to match the characteristics of the device.

If your software does not perform the discovery and initialization sequence, the software must include an alternative method to determine the correct value of the device-dependent registers. The HPS boot ROM code enables discovery and initialization by default (that is, `bootstrap_inhibit_init = 0`).

Local Memory Buffer

The NAND flash controller has three local SRAM memory buffers. Of the three, only the read and write clocks are asynchronous.

- **ECC buffer**—Operates at the core clock, it enables the simultaneous process between sending out the data to the host and receiving data from the device to achieve a line rate operation.
- **Write FIFO**—The write port operates at `ac_lk` and the read port operates at `clk`. The buffer must be large enough to pre-allocate all the read data associated with the number of the outstanding read requests that it could issue to the hosts. Up to 8 outstanding read requests are supported, with a maximum burst size of 64 bytes. Therefore, the write FIFO buffer is a 128×32 -bit memory (512 total bytes).
- **Read FIFO**—The read port operates at `ac_lk` and the write port operates at `clk`. Regardless of the number of outstanding read requests that it could receive from the host, it only requires a buffer size that is just enough to sustain the streaming from the device memory to the system bus. Therefore, the read FIFO buffer is a 32×32 -bit memory (128 total bytes). Since the size of the read FIFO is relatively small for SRAM implementation, it can be implemented using flops.

Clocks

To minimize the number of clocks, Clock Manager outputs software managed enables to the USB, SPI Masters, QSPI and NAND peripherals. The software enable for NAND is `nand_clk_en` and is set to ENABLE by default. Also, in Boot Mode, `nand_clk_en` is active to ensure that all clocks are active if RAM is cleared for security.

Table 13-4: Clock Inputs to NAND Flash Controller

Clock Signal	Description
<code>nand_x_clk</code>	Clock for master and slave interfaces and the ECC sector buffer.
<code>nand_clk</code>	Clock for the NAND flash controller.

The frequency of `nand_x_clk` is four times the frequency of `nand_clk`.

Clock Generation

The clock manager sends the top level clock from the HPS.

The clock manager sends the 200 MHz clock, `l4_mp_clk`, to the NAND Flash Controller. This clock becomes the NAND reference clock called `nand_mp_clk`. The `nand_mp_clk` is divided by four and is used for input and output. Since the NAND places a 200 MHz limit on the clock, each of these four generated clocks are 50 MHz and called `nand_clk`.

Clock Enable

The `nand_mp_clk` and `nand_clk` clocks have enables.

Clock Switching

When you use clock switching, you must follow the following requirements:

- Ensure that there is no activity.
- Software must disable this module during the frequency switch and re-enable it after the frequency has changed.
- When clock switching is complete, the software must reconfigure the NAND initialization registers according to the new frequency before triggering any new transactions onto the flash interface.

Resets

The NAND flash controller has one reset signal, `nand_flash_rst_n`. The reset manager drives this signal to the NAND flash controller on a cold or warm reset.

Related Information

[Reset Manager](#) on page 4-1

Taking the NAND Flash Controller Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

Related Information

- [Module Reset Signals](#) on page 4-5
- [Modules Requiring Software Deassert](#) on page 4-9

Indexed Addressing

The NAND flash controller uses indexed addressing to reduce the address span consumed by the flash controller.

Indirect addressing is implemented by two registers, accessed through the `nanddata` region, as described in the "Register Map for Indexed Addressing" section.

Register Map for Indexed Addressing

Indexed addressing uses registers in the `nanddata` region of the HPS memory map. The `nanddata` region consists of a control register and a variable-size register that allows direct access to flash memory, as detailed in the following table.

Table 13-5: Register Map for Indexed Addressing

Register Name	Offset Address	Usage
Control	0x0	Identifies the page of flash memory to be read or written. Software writes the 32-bit control information consisting of map command type, block, and page address. The upper four bits must be set to 0. For specific usage of the Control register, refer to "Command Mapping".
Data	0x10	The Data register is a page-size window into the NAND flash. By reading from or writing to locations starting at this offset, the software reads directly from or writes directly to the page and block of NAND flash memory specified by the Control register. The Data register is always addressed on 32-bit word boundaries, although the physical flash device has an 8-bit-wide data path.

Related Information

- [Command Mapping](#) on page 13-8
- [HPS Peripheral Region Address Map](#) on page 2-17

Indexed Addressing Host Usage

The host uses indexed addressing as follows:

1. Program the 32-bit index-address field into the Control register in the nanddata region. This action provides the flash address parameters to the NAND flash controller.
2. Perform a 32-bit read or write in the Data register.
3. Perform additional 32-bit reads and writes if they are in the same page and block in flash memory.

It is unnecessary to write to the control register for every data transfer if a group of data transfers targets the same page and block address. For example, you can write the control register at the beginning of a page with the block and page address, and then read or write the entire page by directing consecutive transactions to the Data register.

Command Mapping

The NAND flash controller supports several flash controller-specific MAP commands, providing an abstraction level for programming a NAND flash device. By using the MAP commands, you can avoid directly programming device-specific commands. Using this abstraction layer provides enhanced performance. Commands take multiple cycles to send off-chip. The MAP commands let you initiate commands and let the flash controller sequence them off-chip to the NAND device.

The NAND flash controller supports the following flash controller-specific MAP commands:

- MAP00 commands—boot-read or buffer read/write during read-modify-write operations
- MAP01 commands—memory arrays read/write
- MAP10 commands—NAND flash controller commands
- MAP11 commands—low-level direct access

MAP00 Commands

MAP00 commands access a page buffer in the NAND flash device. Addressing always begins at 0x0 and extends to the page size specified by the `device_main_area_size` and `device_spare_area_size` registers in the `config` group. You can use this command to perform a boot read. Use MAP00 commands in read-modify-write (RMW) operations to read or write any word in the buffer. MAP00 commands allow a direct data path to the page buffer in the device.

The host can access the page buffer directly using the MAP00 commands only if there are no other MAP01 or MAP10 commands active on the NAND flash controller.

MAP00 Command Format

Table 13-6: MAP00 Command Format with Address Mapping

The following table shows the format of a MAP00 command. This command is written to the Command register in the `nanddata` region.

Address Bits	Name	Description
31:28	(reserved)	Set to 0
27:26	CMD_MAP	Set to 0
25:13	(reserved)	Set to 0
12:2	BUFF_ADDR	Data width-aligned buffer address on the memory device. Maximum page access is 8 KB.
1:0	(reserved)	Set to 0

MAP00 Usage Limitations

The usage of these commands under normal operations is limited to the following situations:

- They can be used to perform an Execute-in-Place (XIP) boot from the device; reading directly from the page buffer while booting directly from the device.
- MAP00 commands can be used to perform RMW operations where MAP00 writes are used to modify a read page in the device page buffer. Because the NAND flash controller does not perform ECC correction during such an operation, Intel does not recommend this method in an MLC device.
- In association with MAP11 commands, MAP00 commands provide a way for the host to directly access the device bypassing the hardware abstractions provided by NAND flash controller with MAP01 and MAP10 commands. This method is also used for debugging, or for issuing an operation that the flash controller might not support with MAP01 or MAP10 commands.

Restrictions:

- MAP00 commands cannot be used with MAP01 commands to read part of a page. Accesses using MAP01 commands must perform a complete page transfer.
- No ECC is performed during a MAP00 data access.
- DMA must be disabled (the `flag` bit of the `dma_enable` register in the `dma` group must be set to 0) while performing MAP00 operations.

MAP01 Commands

MAP01 commands transfer complete pages between the host memory and a specific page of the NAND flash device. Because the MAP01 commands support only page addresses, the entire page must be read or written at once. The actual number of commands required depends on the size of the data transfer. The Command register points to the first page and block in the transfer. You do not change the Command register when you initiate subsequent transactions in the transfer, but only when the entire page is transferred.

When the NAND flash controller receives a read command, it issues a load operation on the device, waits for the load to complete, and then returns read data. Read data must be read from the start of the page to the end of the page.

Write data must be written from the start of the page to the end of the page. When the NAND flash controller receives confirmation of the transfer, it issues commands to program the data into the device.

The flash controller ignores the byte enables for read and write commands and transfers the entire data width.

MAP01 Command Format

Table 13-7: MAP01 Command Format with Address Mapping

The following table shows the format of a MAP01 command. This command is written to the Command register in the nanddata region.

Address Bits	Name	Description
31:28	(reserved)	Set to 0
27:26	CMD_MAP	Set to 1
25:24	(reserved)	Set to 0
23:<M>	BLK_ADDR	Block address in the device
(<M>-1):0	PAGE_ADDR	Page address in the device

Note: <M> depends on the number of pages per block in the device. $\text{<M>} = \text{ceil}(\log_2(\text{device pages per block}))$. Therefore, use the following values:

- 32 pages per block: <M>=5
- 64 pages per block: <M>=6
- 128 pages per block: <M>=7
- 256 pages per block: <M>=8
- 384 pages per block: <M>=9
- 512 pages per block: <M>=9

MAP01 Usage Limitations

Use the MAP01 command as follows:

- A complete page must be read or written using a MAP01 command. During such transfers, every transaction from the host must have the same block and page address. The NAND flash controller internally keeps track of how much data it reads or writes.
- MAP00 commands cannot be used in between using MAP01 commands for reading or writing a page.
- DMA must be disabled (the `flag` bit of the `dma_enable` register in the `dma` group must be set to 0) while the host is performing MAP01 operations directly. If the host issues MAP01 commands to the NAND flash controller while DMA is enabled, the flash controller discards the request and generates an `unsp_cmd` interrupt.

MAP10 Commands

MAP10 commands provide an interface to the control plane of the NAND flash controller. MAP10 commands control special functions of the flash device, such as erase, lock, unlock, copy back, and page spare area access. Data passed in this command pathway targets the NAND flash controller rather than the flash device. Unlike other command types, the data (input or output) related to these transactions does not affect the contents of the flash device. Rather, this data specifies and performs the exact commands of the flash controller. Only the lower 16 bits of the Data register contain the relevant information.

MAP10 Command Format

Table 13-8: MAP10 Command Format with Address Mapping

The following table shows the format of a MAP10 command. This command is written to the Command register in the `nanddata` region.

Address Bits	Name	Description
31:28	(reserved)	Set to 0
27:26	CMD_MAP	Set to 2
25:24	(reserved)	Set to 0
23:<M>	BLK_ADDR	Block address in the device
(<M>-1):0	PAGE_ADDR	Page address in the device

Note: <M> depends on the number of pages per block in the device, as follows:

- 32 pages per block: <M>=5
- 64 pages per block: <M>=6
- 128 pages per block: <M>=7
- 256 pages per block: <M>=8
- 384 pages per block: <M>=9
- 512 pages per block: <M>=9

MAP10 Operations**Table 13-9: MAP10 Operations**

Command	Function
0x01	Sets block address for erase and initiates operation
0x10	Sets unlock start address
0x11	Sets unlock end address and initiates unlock
0x21	Initiates a lock of all blocks
0x31	Initiates a lock-tight of all blocks
0x41	Sets up for spare area access
0x42	Sets up for default area access
0x43	Sets up for main+spare area access
0x60	Loads page to the buffer for a RMW operation
0x61	Sets the destination address for the page buffer in RMW operation
0x62	Writes the page buffer for a RMW operation
0x1000	Sets copy source address
0x11<PP>	Sets copy destination address and initiates a copy of <PP> pages
0x20<PP>	Sets up a pipeline read-ahead of <PP> pages
0x21<PP>	Sets up a pipeline write of <PP> pages

MAP10 Usage Limitations

Use the MAP10 commands as follows:

- MAP10 commands should be used to issue commands to the controller, such as erase, copy-back, lock, or unlock.
- MAP10 pipeline commands should also be used to read or write consecutive multiple pages from the flash device within a device block boundary. The host must first issue a MAP10 pipeline read or write command and then issue MAP01 commands to do the actual data transfers. The MAP10 pipeline read or write command instructs the NAND flash controller to use high-performance commands such as cache or multiplane because the flash controller has knowledge of multiple consecutive pages to be read. The pages must not cross a block boundary. If a block boundary is crossed, the flash controller generates an unsupported command (unsup_cmd) interrupt and drops the command.
- Up to four pipeline read or write commands, at the same time, can be issued to the NAND flash controller.
- While the NAND flash controller is performing MAP10 pipeline read or write commands, DMA must be disabled (the `flag` bit of the `dma_enable` register in the `dma` group must be set to 0). DMA must be disabled because the host is directly transferring data from and to the flash device through the flash controller.

MAP11 Commands

MAP11 commands provide direct access to the NAND flash controller's address and control cycles, allowing software to issue the commands directly to the flash device using the Command and Data registers. The MAP11 command is useful if the flash device supports a device-specific command not included with standard flash commands. It can also be useful for low-level debugging.

MAP11 commands provide a direct control path to the flash device. These commands execute command, address, and data read and write cycles directly on the NAND device interface. The host can issue only single-beat accesses to the `nanddata` region while using MAP11 commands. The following are the usage requirements:

- Command, address, and write data values are placed in the Data register.
- Command and address cycles to the device must be a write transaction on the host bus.
- For data cycles, the type of transaction on the host bus (read/write) determines the data cycle type on the device interface.
- On a read, the returned data also appears in the Data register.
- The Control register encodes the control operation type.

MAP11 Control Format

Table 13-10: MAP11 Control Format with Address Mapping

The following table shows the format of a MAP11 command. This command is written to the Command register in the `nanddata` region.

Address Bits	Name	Description
31:28	(reserved)	Set to 0
27:26	CMD_MAP	Set to 3
25:2	(reserved)	Set to 0

Address Bits	Name	Description
1:0	TYPE	Sets the control type as follows: • 0 = Command cycle • 1 = Address cycle • 2 = Data Read/Write Cycle

MAP11 Usage Limitations

Use the MAP11 commands as follows:

- Use MAP11 commands only in special cases, for debugging or sending device-specific commands that are not supported by the NAND flash controller.
- DMA must be disabled before you use MAP11 operations.
- The host can use only single beat access transfers when using MAP11 commands.

Note: MAP11 commands provide direct, unstructured access to the NAND flash device. Incorrect use can lead to unpredictable behavior.

Data DMA

The DMA transfers data with minimal host involvement. Software initiates data DMA with the MAP10 command.

The `flag` bit of the `dma_enable` register in the `dma` group enables data DMA functionality. Only enable or disable this functionality when there are no active transactions pending in the NAND flash controller. When the DMA is enabled, the flash controller initiates one DMA transfer per MAP10 command over the DMA master interface. When the DMA is disabled, all operations with the flash controller occur through the memory-mapped `nanddata` region.

The NAND flash controller supports up to four outstanding DMA commands, and ignores additional DMA commands. If software issues more than four outstanding DMA commands, the flash controller issues the `unsup_cmd` interrupt. On receipt of a DMA command, the flash controller performs command sequencing to transfer the number of pages requested in the DMA command. The DMA master reads or writes page data from the system memory in programmed burst-length chunks. After the DMA command completes, the flash controller issues an interrupt, and starts working on the next queued DMA command.

Pipelining allows the NAND flash controller to optimize its performance while executing back-to-back commands of the same type.

With certain restrictions, non-DMA MAP10 commands can be issued to the NAND flash controller while the flash controller is servicing DMA transactions. MAP00, MAP01, and MAP11 commands cannot be issued while DMA mode is enabled because the flash controller is operating in an extremely tightly-coupled, high-performance data transfer mode. On receipt of erroneous commands (MAP00, MAP01 or MAP11), the flash controller issues an `unsup_cmd` interrupt to inform the host about the violating command.

Consider the following points when using the DMA:

- A data DMA command is a type of MAP10 command. This command is interpreted by the data DMA engine and not by the flash controller core.
- No MAP01, MAP00, or MAP11 commands are allowed when DMA is enabled.
- Before the flash controller can accept data DMA commands, DMA must be enabled by setting the `flag` bit of the `dma_enable` register in the `dma` group.
- When DMA is enabled and the DMA engine initiates data transfers, ECC can be enabled for as-needed data correction concurrent with the data transfer.
- MAP10 commands are used along with data movements similar to MAP01 commands.
- With the exception of data DMA commands and MAP10 pipeline read and write commands, all other MAP10 commands such as erase, lock, unlock, and copy-back are forwarded to the flash controller.
- At any time, up to four outstanding data DMA commands can be handled by flash controller. During multi-page operations, the DMA transfer must not cross a flash block boundary. If it does, the flash controller generates an unsupported command (`unsup_cmd`) interrupt and drops the command.
- Data DMA commands are typically multi-page read and write commands with an associated pointer in host memory. The multi-page data is transferred to or from the host memory starting from the host memory pointer.
- Data DMA uses the `flash_burst_length` register in the `dma` group to determine the burst length value to drive on the interconnect. The data DMA hardware does not account for the interconnect's boundary crossing restrictions. The host must initialize the starting host address so that the DMA master burst transaction does not cross a 4 KB boundary.

There are two methods for initiating a DMA transaction: the multitransaction DMA command, and the burst DMA command.

Multi-Transaction DMA Command

The NAND flash controller processes multitransaction DMA commands only if it receives all four command-data pairs in order. The flash controller responds to out-of-order commands with an `unsup_cmd` interrupt. The flash controller also responds with an `unsup_cmd` interrupt if sequenced commands are interleaved with other flash controller MAP commands.

To initiate DMA with a multitransaction DMA command, you send four command-data pairs to the NAND flash controller through the Control and Data registers in the `nanddata` region, as shown in "Command-Data Pair Formats".

Related Information

[Command-Data Pair Formats](#) on page 13-16

Command-Data Pair Formats

Table 13-11: Command-Data Pair 1

	31:28	27:26	25:24	23:<M>		(<M> – 1):0
Command	0x0	0x2	0x0	Block address		Page address
Note: <M> = ceil(log2(<device pages per block>)). Therefore, use the following values:						
• 32 pages per block: <M>=5 • 64 pages per block: <M>=6 • 128 pages per block: <M>=7 • 256 pages per block: <M>=8 • 384 pages per block: <M>=9 • 512 pages per block: <M>=9						
	31:16			15:12	11:8	7:0
Data	0x0			0x2	0x0 = Read 0x1 = Write	<PP>= Number of pages

Table 13-12: Command-Data Pair 2

	31:28	27:26	25:24	23:8		7:0
Command	0x0	0x2	0x0	Memory address high		0x0
	31:16			15:12	11:8	7:0
Data	0x0			0x2	0x2	0x0

Table 13-13: Command-Data Pair 3

	31:28	27:26	25:24	23:8	7:0	
Command	0x0	0x2	0x0	Memory address low ⁽³⁰⁾	0x0	
	31:16			15:12	11:8	7:0
Data	0x0			0x2	0x3	0x0

Table 13-14: Command-Data Pair 4

	31:28	27:26	25:24	23:17	16	15:8	7:0
Command	0x0	0x2	0x0	0x0	IN T ⁽³¹⁾	Burst length ⁽³²⁾	0x0

⁽²⁹⁾ <M> depends on the number of pages per block in the device. For more information about <M>, see the Note at the bottom of this table.

⁽³⁰⁾ The buffer address in host memory, which must be aligned to 32 bits.

	31:28	27:26	25:24	23:17	16	15:8	7:0	
Note: INT controls the value of the dma_cmd_comp bit of the intr_status0 register in the status group at the end of the DMA transfer. INT can take on one of the following values: • 0—Do not interrupt host. The dma_cmd_comp bit is set to 0. • 1—Interrupt host. The dma_cmd_comp bit is set to 1.								
	31:16					15:12	11:8	7:0
Data	0x0					0x2	0x4	0x0

Related Information

- [Indexed Addressing](#) on page 13-7
- [Burst DMA Command](#) on page 13-17

Using Multi-Transaction DMA Commands

If you want the NAND flash controller DMA to perform cacheable accesses then you must configure the cache bits by writing the l3master register in the nandgrp group in the system manager. The NAND flash controller DMA must be idle before you use the system manager to change its cache capabilities.

You can issue non-DMA MAP10 commands while the NAND flash controller is in DMA mode. For example, you might trigger a host-initiated page move between DMA commands, to achieve wear leveling. However, do not interleave non-DMA MAP10 commands between the command-data pairs in a set of multitransaction DMA commands. You must issue all four command-data pairs shown in the above tables before sending a different command.

Note: Do not issue MAP00, MAP01 or MAP11 commands while DMA is enabled.

MAP10 commands in multitransaction format are written to the Data register at offset 0x10 in nanddata, the same as MAP10 commands in increment four (INCR4) format (described in "Burst DMA Command").

Related Information

- [Indexed Addressing](#) on page 13-7
- [Burst DMA Command](#) on page 13-17
- [System Manager](#) on page 6-1

Burst DMA Command

You can initiate a DMA transfer by sending a command to the NAND flash controller as a burst transaction of four 16-bit accesses. This form of DMA command might be useful for initiating DMA transfers from custom IP in the FPGA fabric. Most processor cores cannot use this form of DMA command, because they cannot control the width of the burst.

When DMA is enabled, the NAND flash controller recognizes the MAP10 pipeline DMA command as an INCR4 command, in the format shown in the following table. The address decoding for MAP10 pipeline DMA command remains the same, as shown in "MAP10 Command Format".

⁽³¹⁾ INT specifies the host interrupt to be generated at the end of the complete DMA transfer. For more information about INT, see the Note at the bottom of this table.

⁽³²⁾ Can be only 4, 16, 32, or 64 bytes. No other values are valid.

MAP10 commands in INCR4 format are written to the Data register at offset 0x10 in `nanddata`, the same as MAP10 commands in multitransaction format (described in the "Multi-Transaction DMA Command").

Table 13-15: MAP10 Burst DMA (INCR4) Command Structure

The following table lists the MAP10 burst DMA command structure. The burst DMA command carries the same information as the multi-transaction DMA command-data pairs, but in a very different format.

Data Beat	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Beat 0	0x2				0x0: read. 0x1: write.				<PP>=number of pages							
Beat 1 ⁽³³⁾	Memory address high															
Beat 2 ⁽³³⁾	Memory address low															
Beat 3	0x0							INT ⁽³⁴⁾	Burst length							

Note: INT controls the value of the `dma_cmd_comp` bit of the `intr_status0` register in the status group at the end of the DMA transfer. INT can take on one of the following values:

0—Do not interrupt host. The `dma_cmd_comp` bit is set to 0.

1—Interrupt host. The `dma_cmd_compp` bit is set to 1.

You can optionally send the 16-bit fields in the above table to the NAND flash controller as four separate bursts of length 1 in sequential order. Intel recommends this method.

If you want the NAND flash controller DMA to perform cacheable accesses, you must configure the cache bits by writing the `l3master` register in the `nandgrp` group in the system manager. The NAND flash controller DMA must be idle before you use the system manager to modify its cache capabilities.

Related Information

- [Multi-Transaction DMA Command](#) on page 13-15
- [MAP10 Command Format](#) on page 13-11
- [System Manager](#) on page 6-1

ECC

The NAND flash controller incorporates ECC logic to calculate and correct bit errors. The flash controller uses a Bose-Chaudhuri-Hocquenghem (BCH) algorithm for detection of multiple errors in a page.

The NAND flash controller supports 512- and 1024-byte ECC sectors. The flash controller inserts ECC check bits for every 512 or 1024 bytes of data, depending on the selected sector size. After 512 or 1024 bytes, the flash controller writes the ECC check bit information to the device page.

ECC information is striped in between 512 or 1024 bytes of data across the page. The NAND flash controller reads ECC information in the same pattern and performs a calculation to check for the presence of errors.

⁽³³⁾ The buffer address in host memory, which must be aligned to 32 bits.

⁽³⁴⁾ INT specifies the host interrupt to be generated at the end of the complete DMA transfer. For more information about INT, see the Note at the bottom of this table.

Correction Capability, Sector Size, and Check Bit Size

Table 13-16: Correction Capability, Sector Size, and Check Bit Size

Correction	Sector Size in Bytes	Check Bit Size in Bytes
4	512	8
8	512	14
16	512	26
24	1024	42

ECC Programming Modes

The NAND flash controller provides the following ECC programming modes that software uses to format a page:

- Main Area Transfer Mode
- Spare Area Transfer Mode
- Main+Spare Area Transfer Mode

Related Information

- [Main Area Transfer Mode](#) on page 13-19
- [Spare Area Transfer Mode](#) on page 13-19
- [Main+Spare Area Transfer Mode](#) on page 13-20

Main Area Transfer Mode

In main area transfer mode, when ECC is enabled, the NAND flash controller inserts ECC check bits in the data stream on writes and strips ECC check bits on reads. Software does not need to manage the ECC sectors when writing a page. ECC checking is performed by the flash controller, so software simply transfers the data.

If ECC is turned off, the NAND flash controller does not read or write ECC check bits.

Figure 13-2: Main Area Transfer Mode for ECC

Sector 0	Sector 1	Sector 2	Sector 3

Spare Area Transfer Mode

The NAND flash controller does not introduce or interpret ECC check bits in spare area transfer mode, and acts as a pass-through for data transfer.

Figure 13-3: Spare Area Transfer Mode for ECC

Sector 3	ECC3	Flags

Main+Spare Area Transfer Mode

In main+spare area transfer mode, the NAND flash controller expects software to format a page as shown in following figure. When ECC is enabled during a write operation, the flash controller-generated ECC check bits replace the ECC check bit data provided by software. During read operations, the flash controller forwards the ECC check bits from the device to the host. If ECC is disabled, page data received from the software is written to the device, and read data received from the device is forwarded to the host.

Figure 13-4: Main+Spare Area Transfer Mode for ECC

Sector 0	ECC0	Sector 1	ECC1	Sector 2	ECC2	Sector 3	ECC3	Flags

Preserving Bad Block Markers

When flash device manufacturers test their devices at the time of manufacture, they mark any bad device blocks that are found. Each bad block is marked at specific, known offsets, typically at the base of the spare area. A bad block marker is any byte value other than 0xFF (the normal state of erased flash).

Bad block markers can be overwritten by the last sector data in a page when ECC is enabled. This happens because the NAND flash controller also uses the main area of a page to store ECC information, which causes the last sector to spill over into the spare area. It is necessary for the system to preserve the bad block information prior to writing data, to ensure the correct identification of bad blocks in the flash device.

You can configure the NAND flash controller to skip over a specified number of bytes when it writes the last sector in a page to the spare area. This option allows the flash controller to preserve bad block markers. To use this option, write the desired offset to the `spare_area_skip_bytes` register in the `config` group. For example, if the device page size is 2 KB, and the device manufacturer stores the bad block markers in the first two bytes in the spare area, set the `spare_area_skip_bytes` register to 2. When the flash controller writes the last sector of the page that overlaps with the spare area, it starts at offset 2 in the spare area, skipping the bad block marker at offset 0. A value of 0 (default) specifies that no bytes are skipped. The value of `spare_area_skip_bytes` must be an even number. For example, if the bad block marker is a single byte, set `spare_area_skip_bytes` to 2.

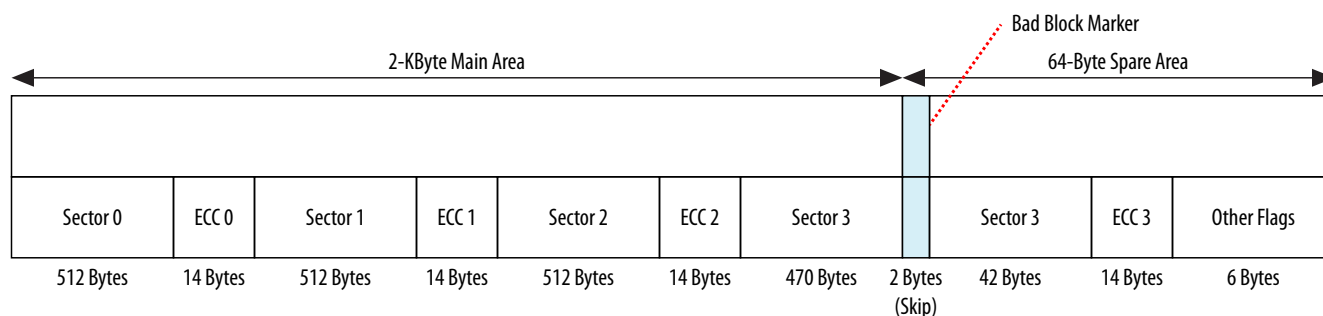
In main area transfer mode, the NAND flash controller does not skip the bad block marker. Instead, it overrides the bad block marker with the value programmed in the `spare_area_marker` register in the `config` group. This 8-bit register is used in conjunction with the `spare_area_skip_bytes` register in the `config` group to determine which bytes in the spare area of a page should be written with a the new marker value. For example, to mark a block as good set the `spare_area_marker` register to 0xFF and set the `spare_area_skip_bytes` register to the number of bytes that the marker should be written to, starting from the base of the spare area.

In the spare area transfer mode, the NAND flash controller ignores the `spare_area_skip_bytes` and `spare_area_marker` registers. The flash controller transfers the data exactly as received from the host or device.

In the main+spare area transfer mode, the NAND flash controller starts writing the last sector in a page into the spare area, starting at the offset specified in the `spare_area_skip_bytes` register. However, the area containing the bad block identifier information is overwritten by the data the host writes into the page. The host writes both the data sectors and the bad block markers. The flash controller depends on the host software to set up the bad block markers properly before writing the data.

Figure 13-5: Bad Block Marker

The following figure shows an example of how the NAND flash controller can skip over a bad block marker. In this example, the flash device has a 2-KB page with a 64-byte spare area. A 14-byte sector ECC is shown, with 8 byte per sector correction.



Related Information

[Transfer Mode Operations](#) on page 13-27

For detailed information about configuring the NAND flash controller for default, spare, or main+spare area transfer mode.

Error Correction Status

The ECC error correction information (ECCorInfo_b01) register, in the ecc group, contains error correction information for each read or write that the NAND flash controller performs. The ECCorInfo_b01 register contains ECC error correction information in the max_errors_b0 and uncor_err_b0 fields.

At the end of data correction for the transaction in progress, ECCorInfo_b01 holds the maximum number of corrections applied to any ECC sector in the transaction. In addition, this register indicates whether the transaction as a whole has correctable errors, uncorrectable errors, or no errors at all. A transaction has no errors when none of the ECC sectors in the transaction has any errors. The transaction is marked as uncorrectable if any one of the sectors is uncorrectable. The transaction is marked as correctable if any one sector has correctable errors and none is uncorrectable.

At the end of each transaction, the host must read this register. The value of this register provides error data to the host about the block. The host can take corrective action after the number of correctable errors encountered reaches a particular threshold value.

NAND Flash Controller Programming Model

This section describes how the NAND flash controller is to be programmed by software running on the microprocessor unit (MPU).

Note: If you write a configuration register and follow it up with a data operation that is dependent on the value of this configuration register, Intel recommends that you read the value of the register before performing the data operation. This read operation ensures that the posted write of the register is completed and takes effect before the data operation is issued to the NAND flash controller.

Basic Flash Programming

This section describes the steps that must be taken by the software to access and control the NAND flash controller.

NAND Flash Controller Optimization Sequence

The software must configure the flash device for interrupt or polling mode, using the `bank0` bit of the `rb_pin_enabled` register in the `config` group. If the device is in polling mode, the software must also program the additional registers, to select the times and frequencies of the polling. Program the following registers in the `config` group:

- Set the `rb_pin_enabled` register to the desired mode of operation for each flash device.
- For polling mode, set the `load_wait_cnt` register to the appropriate value depending on the speed of operation of the NAND flash controller, and the desired wait value.
- For polling mode, set the `program_wait_cnt` register to the appropriate value by software depending on the speed of operation of the NAND flash controller, and the desired wait value.
- For polling mode, set the `erase_wait_cnt` register to the appropriate value by software depending on the speed of operation of the NAND flash controller, and the desired wait value.
- For polling mode, set the `int_mon_cycnt` register to the appropriate value by software depending on the speed of operation of the NAND flash controller, and the desired wait value.

At any time, the software can change any flash device from interrupt mode to polling mode or vice-versa, using the `bank0` bit of the `rb_pin_enabled` register.

The software must ensure that the particular flash device does not have any outstanding transactions before changing the mode of operation for that particular flash device.

Device Initialization Sequence

At initialization, the host software must program the following registers in the `config` group:

- Set the `devices_connected` register to 1.
- Set the `device_width` register to 8.
- Set the `device_main_area_size` register to the appropriate value.
- Set the `device_spare_area_size` register to the appropriate value.
- Set the `pages_per_block` register according to the parameters of the flash device.
- Set the `number_of_planes` register according to the parameters of the flash device.
- If the device allows two ROW address cycles, the `flag` bit of the `two_row_addr_cycles` register must be set to 1. The host program can ensure this condition either of the following ways:
 - Set the `flag` bit of the `bootstrap_two_row_addr_cycles` register to 1 prior to the NAND flash controller's reset initialization sequence, causing the flash controller to initialize the bit automatically.
 - Set the `flag` bit of the `two_row_addr_cycles` register directly to 1.
- Clear the `chip_enable_dont_care` register in the `config` group to 0.

The NAND flash controller can identify the flash device features, allowing you to initialize the flash controller registers to interface correctly with the device, as described in *Discovery and Initialization*.

However, a few NAND devices do not follow any universally accepted identification protocol. If connected to such a device, the NAND flash controller cannot identify it correctly. If you are using such a device, your software must use other means to ensure that the initialization registers are set up correctly.

Related Information

[Discovery and Initialization](#) on page 13-3

Device Operation Control

This section provides a list of registers that you need to program while choosing to use multi-plane or cache operations on the device. If the device does not support multi-plane operations or cache operations, then these registers can be left at their power-on reset values with no impact on the functionality of the NAND flash controller. Even if the device supports these sequences, the software does not need to use them. Software can leave these registers at their power-on reset values.

Program the following registers in the config group to achieve the best performance from a given device:

- Set `flag` bit in the `multiplane_operation` register in the config group to 1 if the device supports multi-plane operations to access the data on the flash device connected to the NAND flash controller. If the flash controller is set up for multi-plane operations, the number of pages to be accessed is always a multiple of the number of planes in the device.
- If the NAND flash controller is configured for multi-plane operation, and if the device has support for multi-plane read command sequence, set the `multiplane_read_enable` register in the config group.
- If the device implements multiplane address restrictions, set the `flag` bit in the `multiplane_addr_restrict` register to 1.
- Initialize the `die_mask` and `first_block_of_next_plane` registers as per device requirements.
- If the device supports cache command sequences, enable the `cache_write_enable` and `cache_read_enable` registers in the config group.
- Clear the `flag` bit of the `copyback_disable` register in the config group to 0 if the device does not support the copyback command sequences. The register defaults to enabled state.
- The `read_mode`, `write_mode` and `copyback_mode` registers, in the config group, currently need not be written by software, because the NAND flash controller is capable of using the correct sequences based on a combination of some multi-plane or cache-related settings of the NAND flash controller and the manufacturer ID. If at some future time these settings change, program the registers to accommodate the change.

ECC Enabling

Before you start any data operation on the flash device, you must decide whether you want the ECC enabled or disabled.

Initialize the memory data before you enable ECC the first time, to prevent spurious ECC.

Follow these steps to enable ECC:

1. Turn on the ECC hardware, but disable the interrupts.
2. Initialize the SRAM in the NAND.
3. Clear the ECC event bits, because these bits may have become asserted after Step 1.
4. Enable the ECC interrupts now that the ECC bits have been set.

If the ECC needs enabling, set up the appropriate correction level depending on the page size and the spare area available on the device.

Set the `flag` bit in the `ecc_enable` register in the config group to 1 to enable ECC. If enabled, the following registers in the config group must be programmed accordingly, else they can be ignored:

- Initialize the `ecc_correction` register to the appropriate correction level.
- Program the `spare_area_skip_bytes` and `spare_area_marker` registers in the config group if the software needs to preserve the bad block marker.

Related Information[ECC](#) on page 13-18**NAND Flash Controller Performance Registers**

These registers specify the size of the bursts on the device interface, which maximizes the overall performance on the NAND flash controller.

Initialize the `flash_burst_length` register in the `dma` group to a value which maximizes the performance of the device interface by minimizing the number of bursts required to transfer a page.

Interrupt and DMA Enabling

Prior to initiating any data operation on the NAND flash controller, the software must set appropriate interrupt status register bits. If the software uses the DMA logic in the flash controller, then the appropriate DMA enable and interrupts bits in the register space must be set.

1. Set the `flag` bit in the `global_int_enable` register in the `config` group to 1, to enable global interrupt.
2. Set the relevant bits of the `intr_en0` register in the `status` group to 1 before initiating any operations if the flash controller is in interrupt mode. Intel recommends that the software reads back this register to ensure clearing an interrupt status. This recommendation applies also to an interrupt service routine.
3. Enable DMA if your application needs DMA mode. Enable DMA by setting the `flag` bit of the `dma_enable` register in the `dma` group. Intel recommends that the software reads back this register to ensure that the mode change is accepted before sending a DMA command to the flash controller.
4. If the DMA is enabled, then set up the appropriate bits of the `dma_intr_en` register in the `dma` group.

Order of Interrupt Status Bits Assertion

The following interrupt status bits, in the `intr_status0` register in the `status` group, are listed in the order of interrupt bit setting:

1. `time_out`—All other interrupt bits are set to 0 when the watchdog `time_out` bit is asserted.
2. `dma_cmd_comp`—This bit signifies the completion of data transfer sequence.⁽³⁵⁾
3. `pipe_cpybck_cmd_comp`—This bit is asserted when a copyback command or the last page of a pipeline command completes.
4. `locked_blk`—This bit is asserted when a program (or erase) is performed on a locked block.
5. `INT_act`—No relationship with other interrupt status bits. Indicates a transition from 0 to 1 on the `ready_busy` pin value for that flash device.
6. `rst_comp`—No relationship with other interrupt status bits. Occurs after a reset command has completed.
7. For an erase command:
 - a. `erase_fail` (if failure)
 - b. `erase_comp`
8. For a program command:
 - a. `locked_blk` (if performed on a locked block)
 - b. `pipe_cmd_err` (if the pipeline sequence is broken by a MAP01 command)
 - c. `page_xfer_inc` (at the end of each page data transfer)
 - d. `program_fail` (if failure)

⁽³⁵⁾ This interrupt status bit is the last to be asserted during a DMA operation to transfer data.

- e. pipe_cpybck_cmd_comp
 - f. program_comp
 - g. dma_cmd_comp (If DMA enabled)
9. For a read command:
- a. pipe_cmd_err (if the pipeline sequence is broken by a MAP01 command)
 - b. page_xfer_inc (at the end of each page data transfer)
 - c. pipe_cpybck_cmd_comp
 - d. load_comp
 - e. ecc_uncor_error (if failure)
 - f. dma_cmd_comp (If DMA enabled)

Timing Registers

You must optimize the following registers for your flash device's speed grade and clock frequency. The NAND flash controller operates correctly with the power-on reset values. However, functioning with power-on reset values is a non-optimal mode that provides loose timing (large margins to the signals).

Set the following registers in the `config` group to optimize the NAND flash controller for the speed grade of the connected device and frequency of operation of the flash controller:

- twhr2_and_we_2_re
- tcwaw_and_addr_2_data
- re_2_we
- acc_clks
- rdwr_en_lo_cnt
- rdwr_en_hi_cnt
- max_rd_delay
- cs_setup_cnt
- re_2_re

Registers to Ignore

You do not need to initialize the following registers in the `config` group:

- The `transfer_spare_reg` register—Data transfer mode can be initialized using MAP10 commands.
- The `write_protect` register—Does not need initializing unless you are testing the write protection feature.

Flash-Related Special Function Operations

This section describes all the special functions that can be performed on the flash memory.

The functions are defined by MAP10 commands as described in *Command Mapping*.

Related Information

[Command Mapping](#) on page 13-8

Erase Operations

Before data can be written to flash, an erase cycle must occur. The NAND flash memory controller supports single block and multi-plane erases.

The controller decodes the block address from the indirect addressing shown in "MAP10 Command Format".

Related Information

MAP10 Command Format on page 13-11

Single Block Erase

A single command is needed to complete a single-block erase, as follows:

1. Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to the desired erase block.
2. Write 0x01 to the Data register.

For a single block erase, the register `multiplane_operation` in the `config` group must be reset.

After the device completes the erase operation, the controller generates an `erase_comp` interrupt. If the erase operation fails, the `erase_fail` interrupt is issued. The failing block's address is updated in the `err_block_addr0` register in the status group.

Multi-Plane Erase

For multi-plane erases, the `number_of_planes` register in the `config` group holds the number of planes in the flash device, and the block address specified must be aligned to the number of planes in the device. The NAND flash controller consecutively erases each block of the memory, up to the number of planes available. Issue this command as follows:

1. Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to the desired erase block.
2. Write 0x01 to the Data register.

For multi-plane erase, the register `multiplane_operation` in the `config` group must be set.

After the device completes erase operation on all planes, the NAND flash controller generates an `erase_comp` interrupt. If the erase operation fails on any of the blocks in a multi-plane erase command, an `erase_fail` interrupt is issued. The failing block's address is updated in the `err_block_addr0` register in the status group.

Lock Operations

The NAND flash controller supports the following features:

- Flash locking—The NAND flash controller supports all flash locking operations.
The flash device itself might have limited support for these functions. If the device does not support locking functions, the flash controller ignores these commands.
- Lock-tight—With the lock-tight feature, the NAND flash controller can prevent lock status from being changed. After the memory is locked tight, the flash controller must be reset before any flash area can be locked or unlocked.

Unlocking a Span of Memory Blocks

To unlock several blocks of memory, perform the following steps:

1. Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to the starting address of the area to unlock.
2. Write 0x10 to the Data register.
3. Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to the ending address of the area to unlock.
4. Write 0x11 to the Data register.

When unlocking a range of blocks, the start block address must be less than the end block address. Otherwise, the NAND flash controller exhibits undetermined behavior.

Locking All Memory Blocks

To lock the entire memory:

1. Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to any memory address.
2. Write 0x21 to the Data register.

Setting Lock-Tight on All Memory Blocks

After the lock-tight is applied, unlocked areas cannot be locked, and locked areas cannot be unlocked. To lock-tight the entire memory:

1. Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to any memory address.
2. Write 0x31 to the Data register.

To disable the lock-tight, reset the memory controller.

Transfer Mode Operations

You can configure the NAND flash controller in one of the following modes of data transfer:

- Default area transfer mode
- Spare area transfer mode
- Main+spare area transfer mode

The NAND flash controller determines the default transfer mode from the setting of transfer_spare_reg register in the config group. Use MAP10 commands to dynamically change the transfer mode from the existing mode to the new mode. All subsequent commands are in the new mode of transfer. You must consider that transfer modes can be changed at logical data transfer boundaries. For example:

- At the beginning or end of a page in case of single page read or write.
- At the beginning or end of a complete multi-page pipeline read or write command.

transfer_spare_reg and MAP10 Transfer Mode Commands

The following table lists the functionality of the MAP10 transfer mode commands, and their mappings to the transfer_spare_reg register in the config group.

Table 13-17: transfer_spare_reg and MAP10 Transfer Mode Commands

transfer_spare_reg	MAP10 Transfer Mode Commands	Resulting NAND Flash Controller Mode
0	0x42	Main ⁽³⁶⁾
0	0x41	Spare

⁽³⁶⁾ Default access mode (0x42) maps to either main (only) or main+spare mode, depending on the value of transfer_spare_reg.

transfer_spare_reg	MAP10 Transfer Mode Commands	Resulting NAND Flash Controller Mode
0	0x43	Main+spare
1	0x42	Main+spare ⁽³⁶⁾
1	0x41	Spare
1	0x43	Main+spare

Related Information

MAP10 Commands on page 13-11

Configure for Default Area Access

You only need to configure for default area access if the transfer mode was previously changed to spare area or main+spare area. To configure default area access:

1. Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to any block.
2. Write 0x42 to the Data register.

The NAND flash controller determines the default area transfer mode from the setting of the transfer_spare_reg register in the config group. If it is set to 1, then the transfer mode becomes main+spare area, otherwise it is main area.

Configure for Spare Area Access

To access only the spare area of the flash device, use the MAP10 command to set up the NAND flash controller to read or write only the spare area on the device. After the flash controller is set up, use MAP01 read and write commands to access the spare area of the appropriate block and page addresses. To configure the NAND flash controller to access the spare area only, perform the following steps:

1. Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to the target block.
2. Write 0x41 to the Data register.

Configure for Main+Spare Area Access

To configure the NAND flash controller to access the main+spare area:

1. Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to the target block.
2. Write 0x43 to the Data register.

Read-Modify-Write Operations

To read a specific page or modify a few words, bytes, or bits in a page, use the RMW operations. A read command copies the desired data from flash memory to a page buffer. You can then modify the information in the buffer using MAP00 buffer read and write commands and issue another command to write that information back to the memory.

The read-modify-write command operates on an entire page. This command is also useful for a copy type operation, where most of a page is saved to a new location. In this type of operation, the NAND flash controller reads the data, modifies a specified number of words in the page, and then writes the modified page to a new location.

Note: Because the data is modified within the page buffer of the flash device, the NAND flash controller ECC hardware is not used in RMW operations. Software must update the ECC during RMW operations.

Note: For a read-modify-write command to work with hardware ECC, the entire page must be read into system memory, modified, then written back to flash without relying on the RMW feature.

Read-Modify-Write Operation Flow

1. Start the flow by reading a page from the memory:

- Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to the starting address of the desired block.
- Write 0x60 to the Data register.

This step makes the page available to you in the page buffer in the flash device.

2. Provide the destination page address:

- Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to the destination address of the desired block.
- Write 0x61 to the Data register.

This step initiates the page program and provides the destination address to the device.

3. Use the MAP00 page buffer read and write commands to modify the data in the page buffer.

4. Write the page buffer data back to memory:

- Write to the command register, setting the CMD_MAP field to 2 and the BLK_ADDR field to the same destination address.
- Write 0x62 to the Data register.

This step performs the write.

After the device completes the load operation, the NAND flash controller issues a load_comp interrupt. A program_comp interrupt is issued when the host issues the write command and the device completes the program operation.

If the page program operation (as a part of an RMW operation) results in a program failure in the device, program_fail interrupt is issued. The failing page's block and page address is updated in the err_block_addr0 and err_page_addr0 registers in the status group.

Copy-Back Operations

The NAND flash controller supports copy back operations. However, the flash device might have limited support for this function. If you attempt to perform a copy-back operation on a device that does not support copy-back, the NAND flash controller triggers an interrupt. An interrupt is also triggered if the source block is not specified before the destination block is specified, or if the destination block is not specified in the next command following a source block specification.

The NAND flash controller cannot do ECC validation in case of copy-back commands. The flash controller copies the ECC data, but does not check it during the copy operation.

Note: Intel recommends that you use copy-back only if the ECC implemented in the flash controller is strong enough so that the next access can correct accumulated errors.

The 8-bit value <PP> specifies the number of pages for copy-back. With this feature, the NAND flash controller can copy multiple consecutive pages with a single command. When you issue a copy-back

command, the flash controller performs the operation in the background. The flash controller puts other commands on hold until the current copy-back completes.

For a multi-plane device, if the `flag` bit in the `multiplane_operation` register in the `config` group is set to 1, multi-plane copy-back is available as an option. In this case, the block address specified must be plane-aligned and the value `<PP>` must specify the total number of pages to copy as a multiple of the number of planes. The block address continues incrementing, keeping the page address fixed, for the total number of planes in the device before incrementing the page address.

A `pipe_cpyback_cmd_comp` interrupt is generated when the flash controller has completed copy-back operation of all `<PP>` pages. If any page program operation (as a part of copy back operation) results in a program failure in the device, the `program_fail` interrupt is issued. The failing page's block and page address is updated in the `err_block_addr0` and `err_page_addr0` registers in the status group.

Copying a Memory Area (Single Plane)

To copy `<PP>` pages from one memory location to another:

1. Write to the command register, setting the `CMD_MAP` field to 2 and the `BLK_ADDR` field to the starting address of the area to be copied.
2. Write 0x1000 to the Data register.
3. Write to the command register, setting the `CMD_MAP` field to 2 and the `BLK_ADDR` field to the starting address of the new area to be written.
4. Write 0x11`<PP>` to the Data register, where `<PP>` is the number of pages to copy.

Copying a Memory Area (Multi-Plane)

To copy `<PP>` pages from one memory location to another:

1. Set the `flag` bit of the `multiplane_operation` register in the `config` group to 1.
2. Write to the command register, setting the `CMD_MAP` field to 2 and the `BLK_ADDR` field to the starting address of the area to be copied. The address must be plane-aligned.
3. Write 0x1000 to the Data register.
4. Write to the command register, setting the `CMD_MAP` field to 2 and the `BLK_ADDR` field to the starting address of the new area to be written. This address must also be plane-aligned.
5. Write 0x11`<PP>` to the Data register, where `<PP>` is the number of pages to copy.

The parameter `<PP>` must be a multiple of the number of planes in the device.

Pipeline Read-Ahead and Write-Ahead Operations

The NAND flash controller supports pipeline read-ahead and write-ahead operations. However, the flash device might have limited support for this function. If the device does not support pipeline read-ahead or write-ahead, the flash controller processes these commands as standard reads or writes.

The NAND flash controller can handle at the most four outstanding pipeline commands, queued up in the order in which the flash controller received the commands. The flash controller operates on the pipeline command at the head of the queue until all the pages corresponding to the pipeline command are executed. The flash controller then pops the pipeline command at the head of the queue and proceeds to work on the next pipeline command in the queue.

Pipeline Read-Ahead Function

The pipeline read-ahead function allows for a continuous reading of the flash memory. On receiving a pipeline read command, the flash controller immediately issues a load command to the device. While data

is read out with MAP01 commands in a consecutive or multi-plane address pattern, the flash controller maintains additional cache or multi-plane read command sequencing for continuous streaming of data from the flash device.

Pipeline read-ahead commands can read data from the queue in this interleaved fashion. The parameter $\langle PP \rangle$ denotes the total number of pages in multiples of the number of planes available, and the block address must be plane-aligned, which keeps the page address constant while incrementing the block address for each page-size chunk of data. After reading from every plane, the NAND flash controller increments the page address and resets the block address to the initial address. You can also use pipeline write-ahead commands in multi-plane mode. The write operation works similarly to the read operation, holding the page address constant while incrementing the block address until all planes are written.

Note: The same four-entry queue is used to queue the address and page count for pipeline read-ahead and write-ahead commands. This commonality requires that you use MAP01 commands to read out all pages for a pipeline read-ahead command before the next pipeline command can be processed. Similarly, you must write to all pages pertaining to pipeline write-ahead command before the next pipeline command can be processed.

Because the value of the `flag` bit of the `multiplane_operation` register in the `config` group determines pipeline read-ahead or write-ahead behavior, it can only be changed when the pipeline registers are empty.

When the host issues a pipeline read-ahead command, and the flash controller is idle, the load operation occurs immediately.

Note: The read-ahead command does not return the data to the host, and the write-ahead command does not write data to the flash address. The NAND flash controller loads the read data. The read data is returned to the host only when the host issues MAP01 commands to read the data. Similarly, the flash controller loads the write data, and writes it to the flash only when the host issues MAP01 commands to write the data.

Set Up a Single Area for Pipeline Read-Ahead

To set up an area for pipeline read-ahead, perform the following steps:

1. Write to the command register, setting the `CMD_MAP` field to 2 and the `BLK_ADDR` field to the starting address of the block to pre-read.
2. Write `0x20<PP>` to the Data register, where the 0 sets this command as a read-ahead and $\langle PP \rangle$ is the number of pages to pre-read. The pages must not cross a block boundary. If a block boundary is crossed, the NAND flash controller generates an unsupported command (`unsup_cmd`) interrupt and drops the command.

The read-ahead command is a hint to the flash device to start loading the next page in the page buffer as soon as the previous page buffer operation has completed. After you set up the read-ahead, use a MAP01 command to actually read the data. In the MAP01 command, specify the same starting address as in the read-ahead.

If the read command received following a pipeline read-ahead request is not to a pre-read page, then an interrupt bit is set to 1 and the pipeline read-ahead or write-ahead registers are cleared. You must issue a new pipeline read-ahead request to re-load the same data. You must use MAP01 commands to read all of the data that is pre-read before the NAND flash controller returns to the idle state.

Pipeline Write-Ahead Function

The pipeline write-ahead function allows for a continuous writing of the flash memory. While data is written with MAP01 commands in a consecutive or multi-plane address pattern, the NAND flash

controller maintains cache or multi-plane command sequences for continuous streaming of data into the flash device.

For pipeline write commands, if any page program results in a failure in the device, a `program_fail` interrupt is issued. The failing page's block and page addresses are updated in the `err_block_addr0` and `err_page_addr0` registers in the status group.

Set Up a Single Area for Pipeline Write-Ahead

To set up an area for pipeline write-ahead:

1. Write to the command register, setting the `CMD_MAP` field to 2 and the `BLK_ADDR` field to the starting address of the block to pre-write.
2. Write `0x21<PP>` to the Data register, where the value 1 sets this command as a write-ahead and `<PP>` is the number of pages to pre-write. The pages must not cross a block boundary. If a block boundary is crossed, the NAND flash controller generates an unsupported command (`unsup_cmd`) interrupt and drops the command.

After you set up the write-ahead, use a `MAP01` command to write the data. In the `MAP01` command, specify the same starting address as in the write-ahead.

If the write command received following a pipeline write-ahead request is not to a pre-written page, then an interrupt bit is set to 1 and the pipeline read-ahead or write-ahead registers are cleared. You must issue a new pipeline write-ahead request to configure the write logic.

You must use `MAP01` commands to write all of the data that is pre-written before the NAND flash controller returns to the idle state.

Other Supported Commands

`MAP01` commands must read or write pages in the same sequence that the pipelined commands were issued to the NAND flash controller. If the host issues multiple pipeline commands, pages must be read or written in the order the pipeline commands were issued. It is not possible to read or write pages for a second pipeline command before completing the first pipeline command. If the pipeline sequence is broken by a `MAP01` command, the `pipe_cmd_err` interrupt is issued, and the flash controller clears the pipeline command queue. The flash controller services the violating incoming `MAP01` read or write request with a normal page read or write sequence.

For a multi-plane device that supports multi-plane programming, you must set the `flag` bit of the `multiplane_operation` register in the `config` group to 1. In this case, the data is interleaved into page-size chunks to consecutive blocks.

A `pipe_cpyback_cmd_comp` interrupt is generated when the NAND flash controller has finished processing a pipeline command and has discarded that command from its queue. At this point of time, the host can send another pipeline command. A pipeline command is popped from the queue, and an interrupt is issued when the flash controller has started processing the last page of pipeline command. Hence, the `pipe_cpyback_cmd_comp` interrupt is issued prior to the last page load in the case of a pipeline read command and start of data transfer of the last page to be programmed, in the case of a pipeline write command.

An additional `program_comp` interrupt is generated when the last page program operation completes in the case of a pipeline write command.

If the device command set requires the NAND flash controller to issue a load command for the last page in the pipeline read command, a `load_comp` interrupt is generated after the last page load operation completes.

The pipeline commands sequence advanced commands in the device, such as cache and multi-plane. When the NAND flash controller receives a multi-page read or write pipeline command, it sequences commands sent to the device depending on settings in the following registers in the `config` group:

- `cache_read_enable`
- `cache_write_enable`
- `multiplane_operation`

For a device that supports cache read sequences, the `flag` bit of the `cache_read_enable` register must be set to 1. The NAND flash controller sequences each multi-page pipeline read command as a cache read sequence. For a device that supports cache program command sequences, `cache_write_enable` must be set. The flash controller sequences each multi-page write pipeline command as a cache write sequence.

For a device that has multi-planes and supports multi-plane program commands, the NAND flash controller register `multiplane_operation`, in the `config` group, must be set. On receiving the multi-page pipeline write command, the flash controller sequences the device with multi-plane program commands and expects that the host transfers data to the flash controller in an even-odd block increment addressing mode.

NAND Flash Controller Address Map and Register Definitions

The address map and register definitions for the NAND Flash Controller consist of the following regions:

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter.
- [Arria V Address Map and Register Definitions](#)
Web-based address map and register definitions



2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) provides a Secure Digital/Multimedia Card (SD/MMC) controller for interfacing to external SD and MMC flash cards, secure digital I/O (SDIO) devices, and Consumer Electronics Advanced Transport Architecture (CE-ATA) hard drives. The SD/MMC controller enables you to store boot images and boot the processor system from the removable flash card. You can also use the flash card to expand the on-board storage capacity for larger applications or user data. Other applications include interfacing to embedded SD (eSD) and embedded MMC (eMMC) non-removable flash devices.

The SD/MMC controller is based on the Synopsys DesignWare Mobile Storage Host (SD/MMC controller) controller.

This document refers to SD/SDIO commands, which are documented in detail in the *Physical Layer Simplified Specification, Version 3.01* and the *SDIO Simplified Specification, Version 2.00* described on the SD Association website.

Related Information

SD Association

To learn more about how SD technology works, visit the SD Association website (www.sdcard.org).

Features of the SD/MMC Controller

The HPS SD/MMC controller offers the following features:

- Supports HPS boot from mobile storage
- Supports the following standards or card types:⁽³⁷⁾
 - SD, including eSD—version 3.0
 - SDIO, including embedded SDIO (eSDIO)—version 3.0
 - CE-ATA—version 1.1
- Supports various types of multimedia cards, MMC version 4.41⁽³⁸⁾
 - MMC: 1-bit data bus
 - Reduced-size MMC (RSMHC): 1-bit and 4-bit data bus
 - MMCPlus: 1-bit, 4-bit, and optional 8-bit data bus
 - MMCmobile: 1-bit data bus
 - Embedded MMC (eMMC): 1-bit, 4-bit, and 8-bit data bus
- Integrated descriptor-based direct memory access (DMA)
- Internal 4 KB receive and transmit FIFO buffer

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

Unsupported Features

- Card Detect is only supported on interfaces routed via the FPGA fabric. The Card Detect Interface signals are not included if the interface is pinned out to HPS I/O.
- The SD/MMC controller does not directly support voltage switching, card interrupts, or back-end power control of eSDIO card devices. However, you can connect these signals to general-purpose I/Os (GPIOs).
- The SD/MMC controller does not contain a reset output as part of the external card interface. To reset the flash card device, consider using a general purpose output pin.

Related Information

- [MMC Support Matrix](#) on page 14-3
For more information on what is supported, refer to the MMC Support Matrix table.
- [Supported Flash Devices for Cyclone V and Arria V SoC](#)
For more information, refer to the supported SD/SDHC/SDXC/MMC/eMMC flash devices section on this page.

SD Card Support Matrix

Table 14-1: SD Card Support Matrix

Device Card Type	Voltages Supported ⁽³⁹⁾		Bus Modes Supported			Bus Speed Modes Supported			
						Default Speed	High Speed	SDR12	SDR25 ⁽⁴⁰⁾
	3.0 V	1.8 V ⁽⁴¹⁾	1 bit	4 bit	8 bit	12.5 MBps 25 MHz	25 MBps 50 MHz	12.5 MBps 25 MHz	25 MBps 50 MHz
SDSC (SD)	√		√			√	√		
SDHC	√	√	√	√		√	√	√	√
SDXC	√	√	√	√		√	√	√	√
eSD	√	√	√	√		√	√	√	√
SDIO	√	√	√	√		√	√	√	√
eSDIO	√	√	√	√	√	√	√	√	√

Note: Card form factors (such as mini and micro) are not enumerated in the above table because they do not impact the card interface functionality.

⁽³⁷⁾ SD and SDIO do not support SDR50, SDR104, and DDR50 modes.

⁽³⁸⁾ DDR timing is optional for MMCPlus, MMCMobile, and eMMC; but not supported for MMC and RSMCMC.

⁽³⁹⁾ Because SD cards initially operate at 3.0 V and can switch to 1.8V after power-up and the BSEL values are constant during the boot process, transceivers are required to support level-shifting and isolation.

⁽⁴⁰⁾ SDR25 speed mode requires 1.8-V signaling. Note that even if a card supports UHS-I modes (for example SDR50, SDR104, DDR50) it can still communicate at the lower speeds (for example SDR12, SDR25).

⁽⁴¹⁾ Where supported, external transceivers are needed to switch the voltage.

MMC Support Matrix

Table 14-2: MMC Support Matrix

Card Device Type	Max Clock Speed (MHz)	Max Data Rate (MBps)	Voltages Supported		Bus Modes Supported			Bus Speed Modes Supported	
			3.3 V	1.8 V	1 bit	4 bit	8 bit	Default Speed	High Speed
MMC	20	2.5	√		√			√	
RSDMMC	20	10	√		√	√		√	√
MMCPlus	50 ⁽⁴²⁾	50	√		√	√	√ ⁽⁴³⁾	√	√
MMCMobile	50	6.5	√	√	√			√	√
eMMC	50	50	√	√	√	√	√	√	√

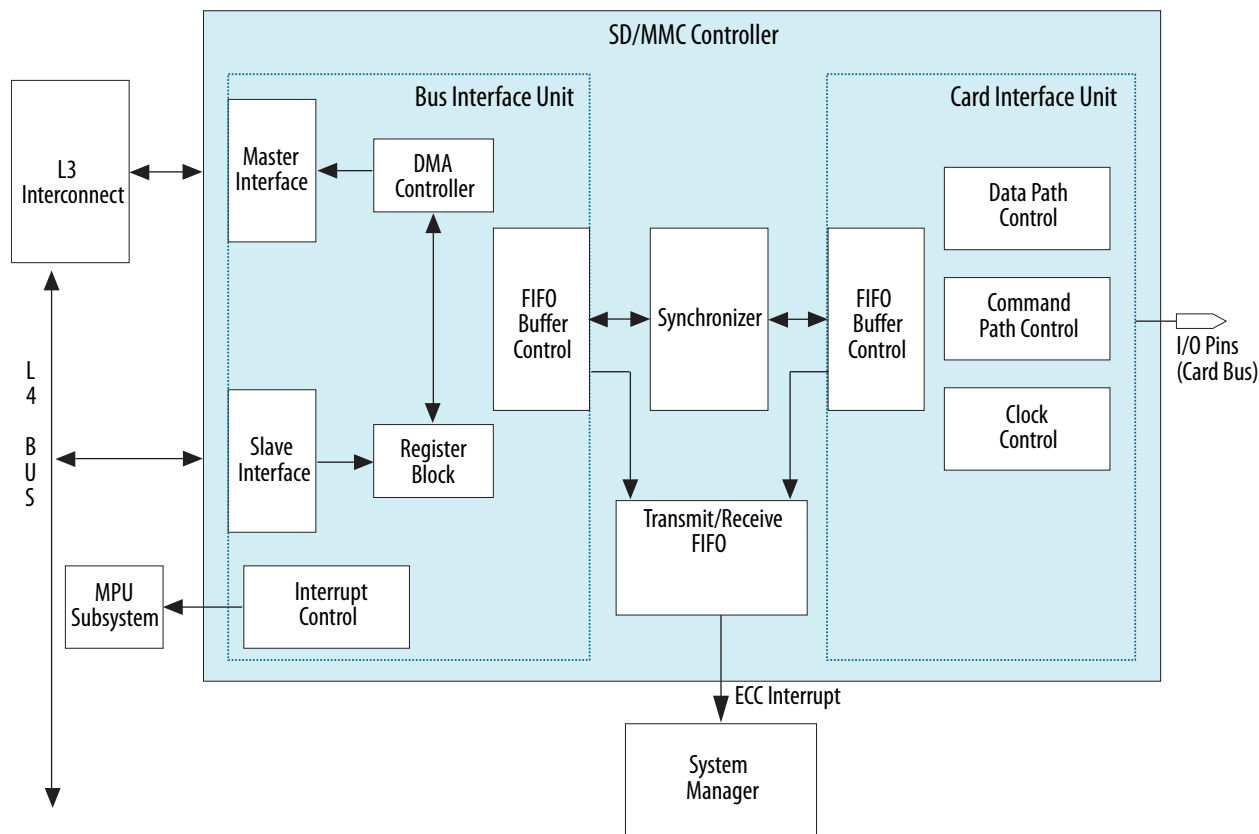
SD/MMC Controller Block Diagram and System Integration

The SD/MMC controller includes a bus interface unit (BIU) and a card interface unit (CIU). The BIU provides a slave interface for a host to access the control and status registers (CSRs). Additionally, this unit also provides independent FIFO buffer access through a DMA interface. The DMA controller is responsible for exchanging data between the system memory and FIFO buffer. The DMA registers are accessible by the host to control the DMA operation. The CIU supports the SD, MMC, and CE-ATA protocols on the controller, and provides clock management through the clock control block. The interrupt control block for generating an interrupt connects to the generic interrupt controller in the Arm Cortex-A9 microprocessor unit (MPU) subsystem.

⁽⁴²⁾ Supports a maximum clock rate of 50 MHz instead of 52 MHz (specified in MMC specification).

⁽⁴³⁾ Optional 8-bit bus mode not supported in all packages.

Figure 14-1: SD/MMC Controller Connectivity



SD/MMC Controller Signal Description

The following table shows the SD/MMC controller signals that are routed to the FPGA and the HPS I/O.

Note: The last five signals in the table are not routed to HPS I/O, but only to the FPGA.

Table 14-3: SD/MMC Controller Interface I/O Pins

Signal	Width	Direction	Description
sdmmc_cclk_out	1	Out	Clock from controller to the card
sdmmc_cmd_i	1	In	Card command
sdmmc_cmd_o	1	Out	
sdmmc_cmd_oe		Out	
sdmmc_pwr_ena_o	1	Out	External device power enable

Signal	Width	Direction	Description
sdmmc_data_i	8	In	Card data
sdmmc_data_o	8	Out	
sdmmc_data_oe	1	Out	
sdmmc_cdn_i	1	In	Card detect signal
sdmmc_wp_i	1	In	Card write protect signal
sdmmc_vs_o	1	Out	Voltage switching between 3.3V and 1.8V
sdmmc_rstn_o	1	Out	Card reset signal used in MMC mode
sdmmc_card_intn_i	1	In	Card interrupt signal

Functional Description of the SD/MMC Controller

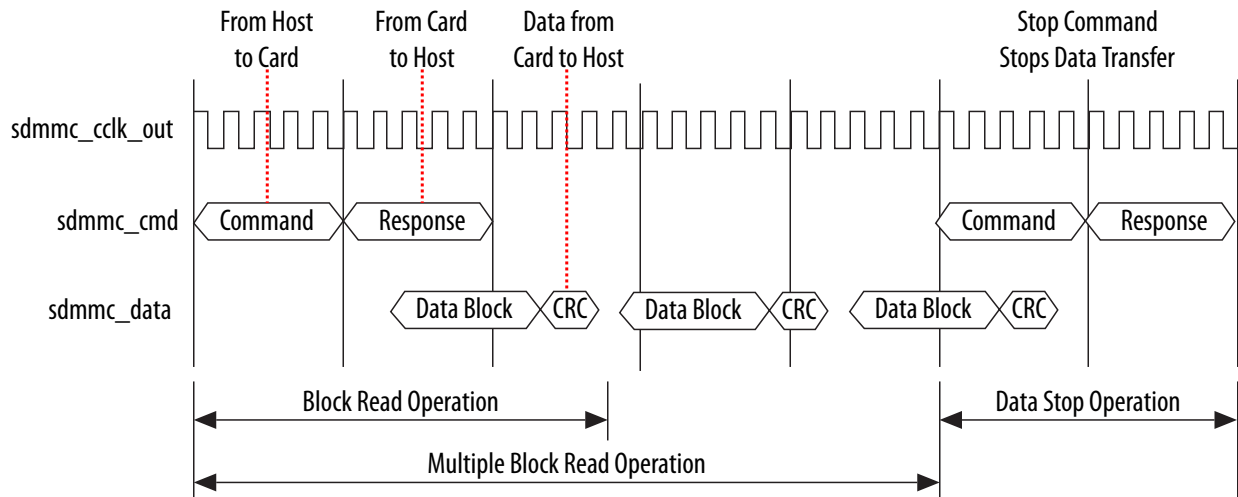
This section describes the SD/MMC controller components and how the controller operates.

SD/MMC/CE-ATA Protocol

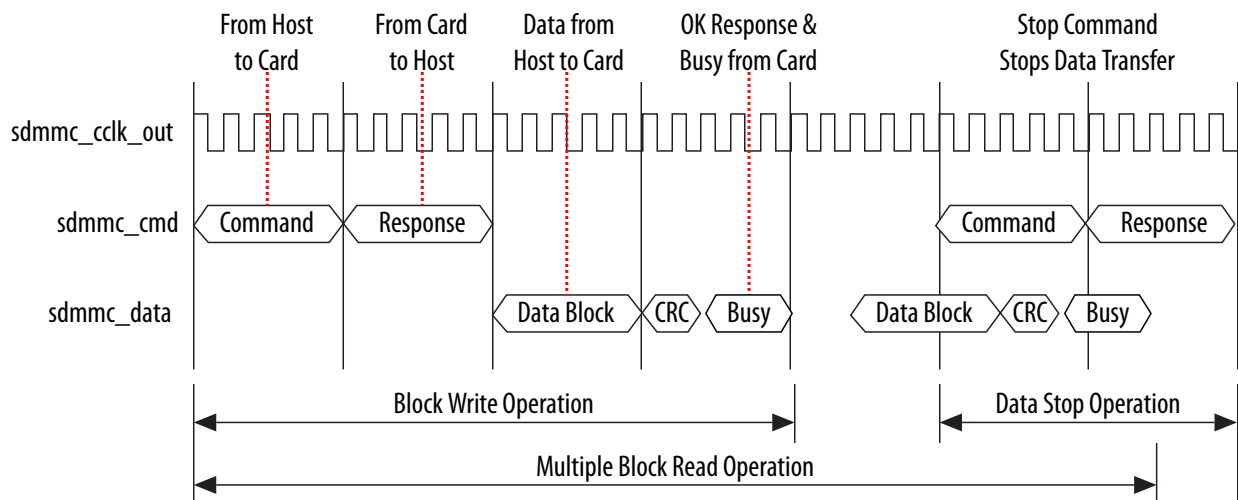
The SD/MMC/CE-ATA protocol is based on command and data bit streams that are initiated by a start bit and terminated by a stop bit. Additionally, the SD/MMC controller provides a reference clock and is the only master interface that can initiate a transaction.[†]

- Command—a token transmitted serially on the CMD pin that starts an operation.[†]
- Response—a token from the card transmitted serially on the CMD pin in response to certain commands.[†]
- Data—transferred serially using the data pins for data movement commands.[†]

In the following figure, the clock is a representative only and does not show the exact number of clock cycles.

Figure 14-2: Multiple-Block Read Operation[†]

The following figure illustrates an example of a command token sent by the host in a multiple-block write operation.

Figure 14-3: Multiple-Block Write Operation[†]

BIU

The Bus Interface Unit (BIU) interfaces with the Card Interface Unit (CIU), and is connected to the level 3 (L3) interconnect and level 4 (L4) peripheral buses. The BIU consists of the following primary functional blocks, which are defined in the following sections:

- Slave interface
- Register block
- FIFO buffer
- Interrupt control
- Internal DMA controller

Slave Interface

The host processor accesses the SD/MMC controller registers and data FIFO buffers through the slave interface.

Register Block

The register block is part of the BIU and provides read and write access to the CSRs.

All registers reside in the BIU clock domain, `l4_mp_clk`. When a command is sent to a card by setting the start command bit (`start_cmd`) of the command register (`cmd`) to 1, all relevant registers needed for the CIU operation are copied to the CIU block. During this time, software must not write to the registers that are transferred from the BIU to the CIU. The software must wait for the hardware to reset the `start_cmd` bit to 0 before writing to these registers again. The register unit has a hardware locking feature to prevent illegal writes to registers.

Registers Locked Out Pending Command Acceptance

After a command start is issued by setting the `start_cmd` bit of the `cmd` register, the following registers cannot be rewritten until the command is accepted by the CIU:[†]

- Command (`cmd`)[†]
- Command argument (`cmdarg`)[†]
- Byte count (`bytcnt`)[†]
- Block size (`blksiz`)[†]
- Clock divider (`clkdiv`)[†]
- Clock enable (`clkena`)[†]
- Clock source (`clksrc`)[†]
- Timeout (`tmout`)[†]
- Card type (`ctype`)[†]

The hardware resets the `start_cmd` bit after the CIU accepts the command. If a host write to any of these registers is attempted during this locked time, the write is ignored and the hardware lock write error bit (`hle`) is set to 1 in the raw interrupt status register (`rintsts`). Additionally, if the interrupt is enabled and not masked for a hardware lock error, an interrupt is sent to the host.[†]

Once a command is accepted, you can send another command to the CIU—which has a one-deep command queue—under the following conditions:[†]

- If the previous command is not a data transfer command, the new command is sent to the SD/MMC/CE-ATA card once the previous command completes.[†]
- If the previous command is a data transfer command and if the wait previous data complete bit (`wait_prvdata_complete`) of the `cmd` register is set to 1 for the new command, the new command is sent to the SD/MMC/CE-ATA card only when the data transfer completes.[†]
- If the `wait_prvdata_complete` bit is 0, the new command is sent to the SD/MMC/CE-ATA card as soon as the previous command is sent. Typically, use this feature to stop or abort a previous data transfer or query the card status in the middle of a data transfer.[†]

Interrupt Controller Unit

The interrupt controller unit generates an interrupt that depends on the `rintsts` register, the interrupt mask register (`intmask`), and the interrupt enable bit (`int_enable`) of the control register (`ctrl`). Once an interrupt condition is detected, the controller sets the corresponding interrupt bit in the `rintsts` register. The bit in the `rintsts` register remains set until the software clears the bit by writing a 1 to the interrupt bit; writing a 0 leaves the bit untouched.

The interrupt controller unit generates active high, level sensitive interrupts that are asserted only when at least one bit in the `rintsts` register is set to 1, the corresponding `intmask` register bit is 1, and the `int_enable` bit of the `ctrl` register is 1.

The `int_enable` bit of the `ctrl` register is cleared during a power-on reset, and the `intmask` register bits are set to 0x0000000, which masks all the interrupts.

Table 14-4: Interrupt Status Register Bits[†]

Bits	Interrupt	Description
16	SDIO Interrupts [†]	Interrupts from SDIO cards. [†]
15	End Bit Error (read)/Write no CRC (EBE) [†]	Error in end-bit during read operation, or no data CRC received during write operation. [†] Note: For MMC CMD19, there may be no CRC status returned by the card. Hence, EBE is set for CMD19. The application should not treat this as an error. [†]
14	Auto Command Done (ACD) [†]	Stop/abort commands automatically sent by card unit and not initiated by host; similar to Command Done (CD) interrupt. [†] Recommendation: Software typically need not enable this for non CE-ATA accesses; Data Transfer Over (DTO) interrupt that comes after this interrupt determines whether data transfer has correctly completed. For CE-ATA accesses, if the software sets <code>send_auto_stop_ccsd</code> bit in the control register, then software should enable this bit. [†]
13	Start Bit Error (SBE)	Error in data start bit when data is read from a card. In 4-bit mode, if all data bits do not have start bit, then this error is set.
12	Hardware Locked write Error (HLE) [†]	During hardware-lock period, write attempted to one of locked registers. [†]

Bits	Interrupt	Description
11	FIFO Underrun/Overrun Error (FRUN) [†]	Host tried to push data when FIFO was full, or host tried to read data when FIFO was empty. Typically this should not happen, except due to error in software. [†] Card unit never pushes data into FIFO when FIFO is full, and pop data when FIFO is empty. [†] If IDMAC (Internal Direct Memory Access Controller) is enabled, FIFO underrun/overrun can occur due to a programming error on MSIZE and watermark values in FIFOTH register; for more information, refer to <i>Internal Direct Memory Access Controller (IDMAC)</i> section in the "Synopsys DesignWare Cores Mobile Storage Host Databook".[†]
10	Data Starvation by Host Timeout (HTO) [†]	To avoid data loss, card clock out (cc1k_out) is stopped if FIFO is empty when writing to card, or FIFO is full when reading from card. Whenever card clock is stopped to avoid data loss, data-starvation timeout counter is started with data-timeout value. This interrupt is set if host does not fill data into FIFO during write to card, or does not read from FIFO during read from card before timeout period. [†] Even after timeout, card clock stays in stopped state, with CIU state machines waiting. It is responsibility of host to push or pop data into FIFO upon interrupt, which automatically restarts cc1k_out and card state machines. [†] Even if host wants to send stop/abort command, it still must ensure to push or pop FIFO so that clock starts in order for stop/abort command to send on cmd signal along with data that is sent or received on data line. [†]

Bits	Interrupt	Description
9	Data Read Timeout (DRTO)/Boot Data Start (BDS) [†]	- In Normal functioning mode: Data read timeout (DRTO) Data timeout occurred. Data Transfer Over (DTO) also set if data timeout occurs.[†] - In Boot Mode: Boot Data Start (BDS) When set, indicates that SD/MMC controller has started to receive boot data from the card. A write to this register with a value of 1 clears this interrupt.[†]
8	Response Timeout (RTO)/ Boot Ack Received (BAR) [†]	- In Normal functioning mode: Response timeout (RTO) Response timeout occurred. Command Done (CD) also set if response timeout occurs. If command involves data transfer and when response times out, no data transfer is attempted by SD/MMC controller.[†] - In Boot Mode: Boot Ack Received (BAR) When expect_boot_ack is set, on reception of a boot acknowledge pattern—0-1-0—this interrupt is asserted. A write to this register with a value of 1 clears this interrupt.[†]
7	Data CRC Error (DCRC) [†]	Received Data CRC does not match with locally-generated CRC in CIU; expected when a negative CRC is received. [†]
6	Response CRC Error (RCRC) [†]	Response CRC does not match with locally-generated CRC in CIU. [†]
5	Receive FIFO Data Request (RXDR) [†]	Interrupt set during read operation from card when FIFO level is greater than Receive-Threshold level.[†] Recommendation: In DMA modes, this interrupt should not be enabled.[†] ISR, in non-DMA mode: pop RX_WMark + 1 data from FIFO [†]

Bits	Interrupt	Description
4	Transmit FIFO Data Request (TXDR) [†]	Interrupt set during write operation to card when FIFO level reaches less than or equal to Transmit-Threshold level.[†] Recommendation: In DMA modes, this interrupt should not be enabled.[†] ISR in non-DMA mode: [†] <pre> if (pending_bytes > \ (FIFO_DEPTH - TX_WMark))[†] push (FIFO_DEPTH - \ TX_WMark) data into FIFO[†] else[†] push pending_bytes data \ into FIFO[†] </pre>
3	Data Transfer (DTO) [†]	Data transfer completed, even if there is Start Bit Error or CRC error. This bit is also set when “read data-timeout” occurs or CCS is sampled from CE-ATA device.[†] Recommendation: In non-DMA mode, when data is read from card, on seeing interrupt, host should read any pending data from FIFO. In DMA mode, DMA controllers guarantee FIFO is flushed before interrupt.[†] Note: DTO bit is set at the end of the last data block, even if the device asserts MMC busy after the last data block.[†]
2	Command Done (CD) [†]	Command sent to card and received response from card, even if Response Error or CRC error occurs. Also set when response timeout occurs or CCSD sent to CE-ATA device.[†]
1	Response Error (RE) [†]	Error in received response set if one of following occurs:[†] • Transmission bit != 0[†] • Command index mismatch[†] • End-bit != 1[†]

Bits	Interrupt	Description
0	Card-Detect (CDT) [†]	When one or more cards inserted or removed, this interrupt occurs. Software should read card-detect register (CDETECT, 0x50) to determine current card status.[†] Recommendation: After power-on and before enabling interrupts, software should read card detect register and store it in memory. When interrupt occurs, it should read card detect register and compare it with value stored in memory to determine which card(s) were removed/inserted. Before exiting ISR, software should update memory with new card-detect value.[†]

Interrupt Setting and Clearing

The SDIO Interrupts, Receive FIFO Data Request, and Transmit FIFO Data Request interrupts are set by level-sensitive interrupt sources. Therefore, the interrupt source must be first cleared before you can reset the interrupt's corresponding bit in the `rintsts` register to 0.[†]

For example, on receiving the Receive FIFO Data Request interrupt, the FIFO buffer must be emptied so that the FIFO buffer count is not greater than the RX watermark, which causes the interrupt to be triggered.[†]

The rest of the interrupts are triggered by single clock-pulse-width sources.[†]

FIFO Buffer

The SD/MMC controller has a 4 KB data FIFO buffer for storing transmit and receive data. The FIFO buffer memory supports error correction codes (ECCs). Both interfaces to the FIFO buffer support single and double bit error injection. The enable and error injection pins are inputs driven by the system manager and the status pins are outputs driven to the MPU subsystem.

The SD/MMC controller provides outputs to notify the system manager when single-bit correctable errors are detected (and corrected), and when double-bit (uncorrectable) errors are detected. The system manager generates an interrupt to the GIC when an ECC error is detected.

Note: Initialization of memory data before enabling ECC prevents spurious ECC interrupts when you enable ECC for the first time.

Related Information

[System Manager](#) on page 6-1

Internal DMA Controller

Internal DMA controller (AHB Master) enables the core to act as a Master on the AHB to transfer data to and from the AHB.

- Supports 32-bit data
- Supports split, retry, and error AHB responses, but does not support wrap
- Configurable for little-endian or big-endian mode
- Allows the selection of AHB burst type through software

The internal DMA controller has a CSR and a single transmit or receive engine, which transfers data from system memory to the card and vice versa. The controller uses a descriptor mechanism to efficiently move data from source to destination with minimal host processor intervention. You can configure the controller to interrupt the host processor in situations such as transmit and receive data transfer completion from the card, as well as other normal or error conditions. The DMA controller and the host driver communicate through a single data structure.[†]

The internal DMA controller transfers the data received from the card to the data buffer in the system memory, and transfers transmit data from the data buffer in the memory to the controller's FIFO buffer. Descriptors that reside in the system memory act as pointers to these buffers.[†]

A data buffer resides in the physical memory space of the system memory and consists of complete or partial data. The buffer status is maintained in the descriptor. Data chaining refers to data that spans multiple data buffers. However, a single descriptor cannot span multiple data buffers.[†]

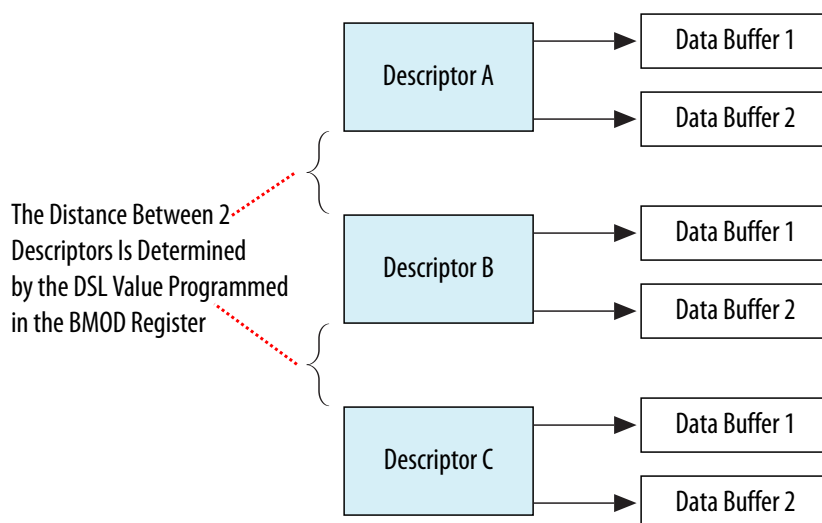
A single descriptor is used for both reception and transmission. The base address of the list is written into the descriptor list base address register (dbaddr). A descriptor list is forward linked. The last descriptor can point back to the first entry to create a ring structure. The descriptor list resides in the physical memory address space of the host. Each descriptor can point to a maximum of two data buffers.[†]

Internal DMA Controller Descriptors

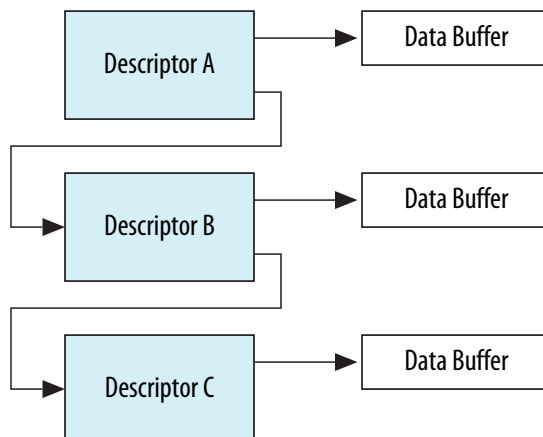
The internal DMA controller uses these types of descriptor structures:[†]

- Dual-buffer structure—The distance between two descriptors is determined by the skip length value written to the descriptor skip length field (dsl) of the bus mode register (bmod).[†]

Figure 14-4: Dual-Buffer Descriptor Structure[†]



- Chain structure—Each descriptor points to a unique buffer, and to the next descriptor in a linked list.[†]

Figure 14-5: Chain Descriptor Structure[†]

Internal DMA Controller Descriptor Address

The descriptor address must be aligned to the 32-bit bus. Each descriptor contains 16 bytes of control and status information.[†]

Table 14-5: Descriptor Format

Name	Off- set	31	30	29:27	26	25:14	13	12:7	6	5	4	3	2	1	0
DES0	0	OWN	CES	—						ER	CH	FS	LD	DIC	—
DES1	4	—				BS2		BS1							
DES2	8	BAP1													
DES3	12	BAP2 or Next Descriptor Address													

Related Information

[Internal DMA Controller Descriptor Fields](#) on page 14-14

Refer to this table for information about each of the bits of the descriptor.

Internal DMA Controller Descriptor Fields

The DES0 field in the internal DMA controller descriptor contains control and status information.

Table 14-6: Internal DMA Controller DES0 Descriptor Field[†]

Bits	Name	Description
31	OWN	When set to 1, this bit indicates that the descriptor is owned by the internal DMA controller. When this bit is set to 0, it indicates that the descriptor is owned by the host. The internal DMA controller resets this bit to 0 when it completes the data transfer.
30	Card Error Summary (CES)	The CES bit indicates whether a transaction error occurred. The CES bit is the logical OR of the following error bits in the <code>rintsts</code> register. • End-bit error (<code>ebe</code>) • Response timeout (<code>rto</code>) • Response CRC (<code>rcrc</code>) • Start-bit error (<code>sbe</code>) • Data read timeout (<code>drto</code>) • Data CRC for receive (<code>dcrc</code>) • Response error (<code>re</code>)
29:6	Reserved	—
5	End of Ring (ER)	When set to 1, this bit indicates that the descriptor list reached its final descriptor. The internal DMA controller returns to the base address of the list, creating a descriptor ring. ER is meaningful for only a dual-buffer descriptor structure.
4	Second Address Chained (CH)	When set to 1, this bit indicates that the second address in the descriptor is the next descriptor address rather than the second buffer address. When this bit is set to 1, BS2 (<code>DES1[25:13]</code>) must be all zeros.
3	First Descriptor (FD)	When set to 1, this bit indicates that this descriptor contains the first buffer of the data. If the size of the first buffer is 0, next descriptor contains the beginning of the data.
2	Last Descriptor (LD)	When set to 1, this bit indicates that the buffers pointed to by this descriptor are the last buffers of the data.
1	Disable Interrupt on Completion (DIC)	When set to 1, this bit prevents the setting of the TI/RI bit of the internal DMA controller status register (<code>idsts</code>) for the data that ends in the buffer pointed to by this descriptor.
0	Reserved	—

Table 14-7: Internal DMA Controller DES1 Descriptor Field[†]

The DES1 descriptor field contains the buffer size.

Bits	Name	Description
31:26	Reserved	—
25:13	Buffer 2 Size (BS2)	This field indicates the second data buffer byte size. The buffer size must be a multiple of four. When the buffer size is not a multiple of four, the resulting behavior is undefined. This field is not valid if DES0[4] is set to 1.
12:0	Buffer 1 Size (BS1)	Indicates the data buffer byte size, which must be a multiple of four bytes. When the buffer size is not a multiple of four, the resulting behavior is undefined. If this field is 0, the DMA ignores the buffer and proceeds to the next descriptor for a chain structure, or to the next buffer for a dual-buffer structure. If there is only one descriptor and only one buffer to be programmed, you need to use only buffer 1 and not buffer 2.

Table 14-8: Internal DMA Controller DES2 Descriptor Field[†]

The DES2 descriptor field contains the address pointer to the data buffer.

Bits	Name	Description
31:0	Buffer Address Pointer 1 (BAP1)	These bits indicate the physical address of the first data buffer. The internal DMA controller ignores DES2 [1:0], because it only performs 32-bit aligned accesses.

Table 14-9: Internal DMA Controller DES3 Descriptor Field[†]

The DES3 descriptor field contains the address pointer to the next descriptor if the present descriptor is not the last descriptor in a chained descriptor structure or the second buffer address for a dual-buffer structure.[†]

Bits	Name	Description
31:0	Buffer Address Pointer 2 (BAP2) or Next Descriptor Address	These bits indicate the physical address of the second buffer when the dual-buffer structure is used. If the Second Address Chained (DES0[4]) bit is set to 1, this address contains the pointer to the physical memory where the next descriptor is present. If this is not the last descriptor, the next descriptor address pointer must be aligned to 32 bits. Bits 1 and 0 are ignored.

Host Bus Burst Access

The internal DMA controller attempts to issue fixed-length burst transfers on the master interface if configured using the fixed burst bit (`fb`) of the `bmod` register. The maximum burst length is indicated and limited by the programmable burst length (`pb1`) field of the `bmod` register. When descriptors are being fetched, the master interface always presents a burst size of four to the interconnect.[†]

The internal DMA controller initiates a data transfer only when sufficient space to accommodate the configured burst is available in the FIFO buffer or the number of bytes to the end of transfer is less than the configured burst-length. When the DMA master interface is configured for fixed-length bursts, it transfers data using the most efficient combination of `INCR4`, `INCR8` or `INCR16` and `SINGLE` transactions. If the DMA master interface is not configured for fixed length bursts, it transfers data using `INCR` (undefined length) and `SINGLE` transactions.[†]

Host Data Buffer Alignment

The transmit and receive data buffers in system memory must be aligned to a 32-bit boundary.

Buffer Size Calculations

The driver knows the amount of data to transmit or receive. For transmitting to the card, the internal DMA controller transfers the exact number of bytes from the FIFO buffer, indicated by the buffer size field of the `DES1` descriptor field.[†]

If a descriptor is not marked as last (with the `LD` bit of the `DES0` field set to 0) then the corresponding buffer(s) of the descriptor are considered full, and the amount of valid data in a buffer is accurately indicated by its buffer size field. If a descriptor is marked as last, the buffer might or might not be full, as indicated by the buffer size in the `DES1` field. The driver is aware of the number of locations that are valid.[†] The driver is expected to ignore the remaining, invalid bytes.

Internal DMA Controller Interrupts

Interrupts can be generated as a result of various events. The `idsts` register contains all the bits that might cause an interrupt. The internal DMA controller interrupt enable register (`idinten`) contains an enable bit for each of the events that can cause an interrupt to occur.[†]

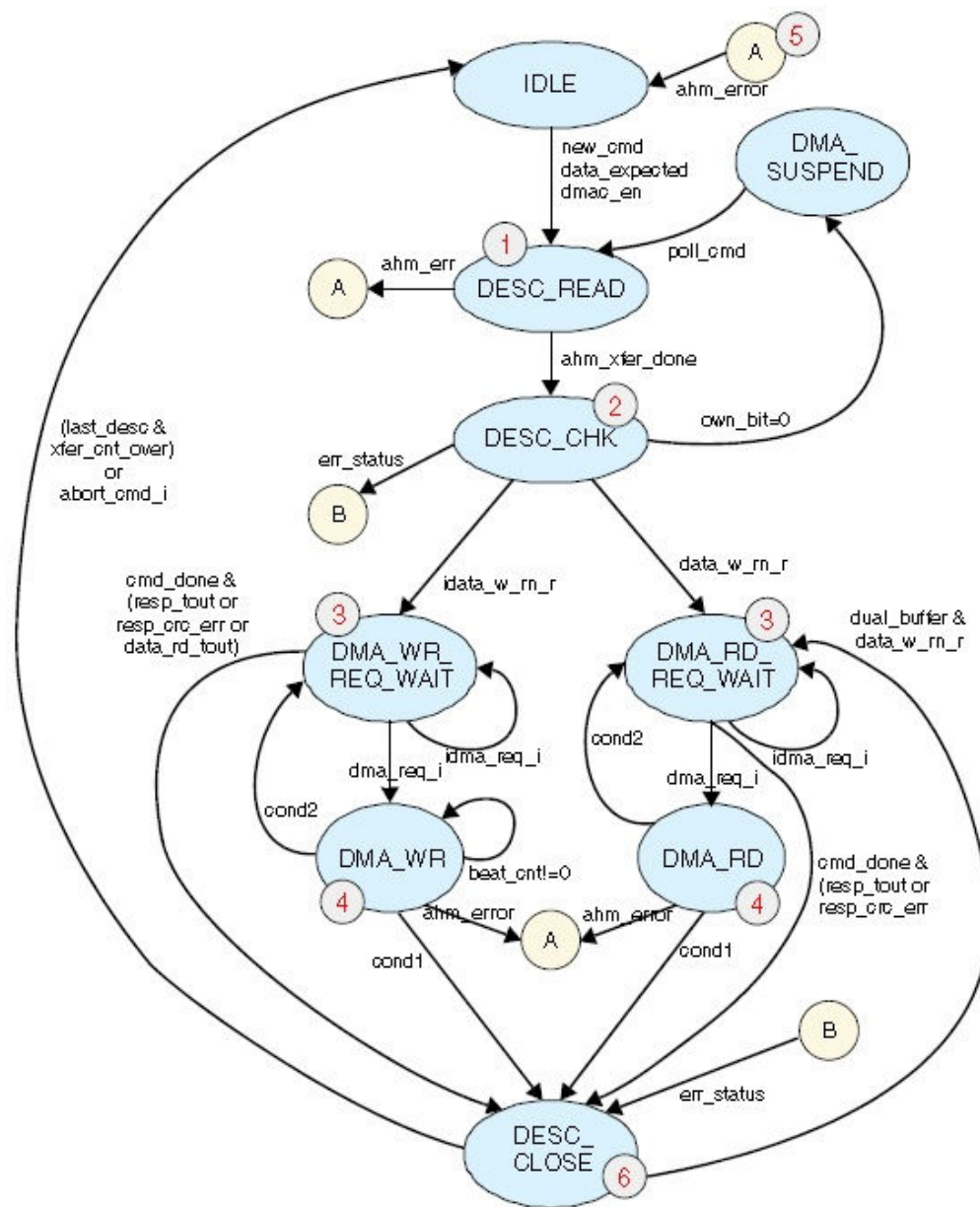
There are two summary interrupts—the normal interrupt summary bit (`nis`) and the abnormal interrupt summary bit (`ais`)—in the `idsts` register.[†] The `nis` bit results from a logical OR of the transmit interrupt (`ti`) and receive interrupt (`ri`) bits in the `idsts` register. The `ais` bit is a logical OR result of the fatal bus error interrupt (`fbe`), descriptor unavailable interrupt (`du`), and card error summary interrupt (`ces`) bits in the `idsts` register.

Interrupts are cleared by writing a 1 to the corresponding bit position.[†] If a 0 is written to an interrupt's bit position, the write is ignored, and does not clear the interrupt. When all the enabled interrupts within a group are cleared, the corresponding summary bit is set to 0. When both the summary bits are set to 0, the interrupt signal is de-asserted.[†]

Interrupts are not queued. If another interrupt event occurs before the driver has responded to the previous interrupt, no additional interrupts are generated. For example, the `ri` bit of the `idsts` register indicates that one or more data has been transferred to the host buffer.[†]

An interrupt is generated only once for simultaneous, multiple events. The driver must scan the `idsts` register for the interrupt cause.[†] The final interrupt signal from the controller is a logical OR of the interrupts from the BIU and internal DMA controller.



Internal DMA Controller Functional State Machine[†]

The following list explains each state of the functional state machine:[†]

1. The internal DMA controller performs four accesses to fetch a descriptor.[†]
2. The DMA controller stores the descriptor information internally. If it is the first descriptor, the controller issues a FIFO buffer reset and waits until the reset is complete.[†]
3. The internal DMA controller checks each bit of the descriptor for the correctness. If bit mismatches are found, the appropriate error bit is set to 1 and the descriptor is closed by setting the OWN bit in the DES0 field to 1.[†]

The `rintsts` register indicates one of the following conditions:[†]

- Response timeout[†]
 - Response CRC error[†]
 - Data receive timeout[†]
 - Response error[†]
4. The DMA waits for the RX watermark to be reached before writing data to system memory, or the TX watermark to be reached before reading data from system memory. The RX watermark represents the number of bytes to be locally stored in the FIFO buffer before the DMA writes to memory. The TX watermark represents the number of free bytes in the local FIFO buffer before the DMA reads data from memory.[†]
 5. If the value of the programmable burst length (PBL) field is larger than the remaining amount of data in the buffer, single transfers are initiated. If dual buffers are being used, and the second buffer contains no data (buffer size = 0), the buffer is skipped and the descriptor is closed.[†]
 6. The OWN bit in descriptor is set to 0 by the internal DMA controller after the data transfer for one descriptor is completed. If the transfer spans more than one descriptor, the DMA controller fetches the next descriptor. If the transfer ends with the current descriptor, the internal DMA controller goes to idle state after setting the ri bit or the ti bit of the idsts register. Depending on the descriptor structure (dual buffer or chained), the appropriate starting address of descriptor is loaded. If it is the second data buffer of dual buffer descriptor, the descriptor is not fetched again.[†]

Abort During Internal DMA Transfer

If the host issues an SD/SDIO STOP_TRANSMISSION command (CMD12) to the card while data transfer is in progress, the internal DMA controller closes the present descriptor after completing the data transfer until a Data Transfer Over (DTO) interrupt is asserted. Once a STOP_TRANSMISSION command is issued, the DMA controller performs single burst transfers.[†]

- For a card write operation, the internal DMA controller keeps writing data to the FIFO buffer after fetching it from the system memory until a DTO interrupt is asserted. This is done to keep the card clock running so that the STOP_TRANSMISSION command is reliably sent to the card.[†]
- For a card read operation, the internal DMA controller keeps reading data from the FIFO buffer and writes to the system memory until a DTO interrupt is generated. This is required because DTO interrupt is not generated until and unless all the FIFO buffer data is emptied.[†]

Note: For a card write abort, only the current descriptor during which a STOP_TRANSMISSION command is issued is closed by the internal DMA controller. The remaining unread descriptors are not closed by the internal DMA controller.[†]

Note: For a card read abort, the internal DMA controller reads the data out of the FIFO buffer and writes them to the corresponding descriptor data buffers. The remaining unread descriptors are not closed.[†]

Fatal Bus Error Scenarios

A fatal bus error occurs when the master interface issues an error response. This error is a system error, so the software driver must not perform any further setup on the controller. The only recovery mechanism from such scenarios is to perform one of the following tasks:[†]

- Issue a reset to the controller through the reset manager.[†]
- Issue a program controller reset by writing to the controller reset bit (controller_reset) of the ctrl register.[†]

FIFO Buffer Overflow and Underflow

During normal data transfer conditions, FIFO buffer overflow and underflow does not occur. However, if there is a programming error, a FIFO buffer overflow or underflow can result. For example, consider the following scenarios.[†]

For transmit:[†]

- PBL=4[†]
- TX watermark = 1[†]

For these programming values, if the FIFO buffer has only one location empty, the DMA attempts to read four words from memory even though there is only one word of storage available. This results in a FIFO Buffer Overflow interrupt.[†]

For receive:[†]

- PBL=4[†]
- RX watermark = 1[†]

For these programming values, if the FIFO buffer has only one location filled, the DMA attempts to write four words, even though only one word is available. This results in a FIFO Buffer Underflow interrupt.[†]

The driver must ensure that the number of bytes to be transferred, as indicated in the descriptor, is a multiple of four bytes. For example, if the `bytcnt` register = 13, the number of bytes indicated in the descriptor must be rounded up to 16 because the length field must always be a multiple of four bytes.[†]

PBL and Watermark Levels

This table shows legal PBL and FIFO buffer watermark values for internal DMA controller data transfer operations.[†]

Table 14-10: PBL and Watermark Levels[†]

PBL (Number of transfers)	TX/RX FIFO Buffer Watermark Value
1	greater than or equal to 1
4	greater than or equal to 4
8	greater than or equal to 8
16	greater than or equal to 16
32	greater than or equal to 32
64	greater than or equal to 64
128	greater than or equal to 128
256	greater than or equal to 256

CIU

The Card Interface Unit (CIU) interfaces with the BIU and SD/MMC cards or devices. The host processor writes command parameters to the SD/MMC controller's BIU control registers and these parameters are then passed to the CIU. Depending on control register values, the CIU generates SD/MMC command and data traffic on the card bus according to the SD/MMC protocol. The control register values also decide whether the command and data traffic is directed to the CE-ATA card, and the SD/MMC controller controls the command and data path accordingly.[†]

The following list describes the CIU operation restrictions:[†]

- After a command is issued, the CIU accepts another command only to check read status or to stop the transfer.[†]
- Only one data transfer command can be issued at a time.[†]
- During an open-ended card write operation, if the card clock is stopped because the FIFO buffer is empty, the software must first fill the data into the FIFO buffer and start the card clock. It can then issue only an SD/SDIO STOP_TRANSMISSION (CMD12) command to the card.[†]
- During an SDIO/COMBO card transfer, if the card function is suspended and the software wants to resume the suspended transfer, it must first reset the FIFO buffer and start the resume command as if it were a new data transfer command.[†]
- When issuing SD/SDIO card reset commands (GO_IDLE_STATE, GO_INACTIVE_STATE or CMD52_reset) while a card data transfer is in progress, the software must set the stop abort command bit (stop_abort_t_cmd) in the cmd register to 1 so that the controller can stop the data transfer after issuing the card reset command.[†]
- If the card clock is stopped because the FIFO buffer is full during a card read, the software must read at least two FIFO buffer locations to start the card clock.[†]
- If CE-ATA card device interrupts are enabled (the nIEN bit is set to 0 in the ATA control register), a new RW_BLK command must not be sent to the same card device if there is a pending RW_BLK command in progress (the RW_BLK command used in this document is the RW_MULTIPLE_BLOCK MMC command defined by the CE-ATA specification). Only the Command Completion Signal Disable (CCSD) command can be sent while waiting for the Command Completion Signal (CCS).[†]
- For the same card device, a new command is allowed for reading status information, if interrupts are disabled in the CE-ATA card (the nIEN bit is set to 1 in the ATA control register).[†]
- Open-ended transfers are not supported for the CE-ATA card devices.[†]
- The send_auto_stop signal is not supported (software must not set the send_auto_stop bit in the cmd register) for CE-ATA transfers.[†]

The CIU consists of the following primary functional blocks:[†]

- Command path[†]
- Data path[†]
- Clock control[†]

Command Path

The command path performs the following functions:[†]

- Load card command parameters[†]
- Send commands to card bus[†]
- Receive responses from card bus[†]
- Send responses to BIU[†]
- Load clock parameters[†]
- Drives the P-bit on command pin[†]

A new command is issued to the controller by writing to the BIU registers and setting the start_cmd bit in the cmd register. The command path loads the new command (command, command argument, timeout) and sends an acknowledgement to the BIU.[†]

After the new command is loaded, the command path state machine sends a command to the card bus—including the internally generated seven-term CRC (CRC-7)—and receives a response, if any. The state machine then sends the received response and signals to the BIU that the command is done, and then waits for eight clock cycles before loading a new command. In CE-ATA data payload transfer (RW_MULTIPLE_BLOCK) commands, if the card device interrupts are enabled (the nIEN bit is set to 0

in the ATA control register), the state machine performs the following actions after receiving the response:[†]

- Does not drive the P-bit; it waits for CCS, decodes and goes back to idle state, and then drives the P-bit.[†]
- If the host wants to send the CCSD command and if eight clock cycles are expired after the response, it sends the CCSD pattern on the command pin.[†]

Load Command Parameters

Commands or responses are loaded in the command path in the following situations:[†]

- New command from BIU—When the BIU sends a new command to the CIU, the `start_cmd` bit is set to 1 in the `cmd` register.[†]
- Internally-generated `send_auto_stop`—When the data path ends, the SD/SDIO STOP command request is loaded.[†]
- Interrupt request (IRQ) response with relative card address (RCA) 0x000—When the command path is waiting for an IRQ response from the MMC and a “send irq response” request is signaled by the BIU, the send IRQ request bit (`send_irq_response`) is set to 1 in the `ctrl` register.[†]

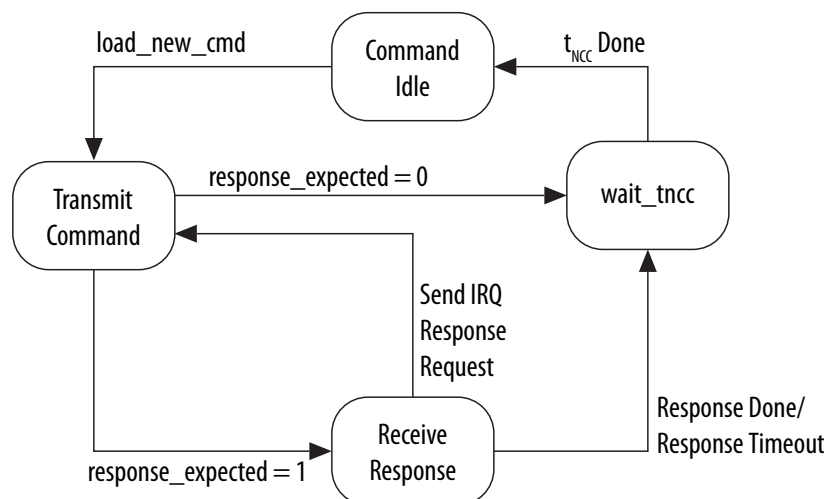
Loading a new command from the BIU in the command path depends on the following `cmd` register bit settings:[†]

- `update_clock_registers_only`—If this bit is set to 1 in the `cmd` register, the command path updates only the `clkena`, `clkdiv`, and `clksrc` registers. If this bit is set to 0, the command path loads the `cmd`, `cmdarg`, and `tmout` registers. It then processes the new command, which is sent to the card.[†]
- `wait_prvdata_complete`—If this bit is set to 1, the command path loads the new command under one of the following conditions:[†]
 - Immediately, if the data path is free (that is, there is no data transfer in progress), or if an open-ended data transfer is in progress (`bytcnt = 0`).[†]
 - After completion of the current data transfer, if a predefined data transfer is in progress.[†]

Send Command and Receive Response

After a new command is loaded in the command path (the `update_clock_registers_only` bit in the `cmd` register is set to 0), the command path state machine sends out a command on the card bus.[†]

Figure 14-6: Command Path State Machine[†]



The command path state machine performs the following functions, according to cmd register bit values:[†]

1. **send_initialization**—Initialization sequence of 80 clock cycles is sent before sending the command.[†]
2. **response_expected**—A response is expected for the command. After the command is sent out, the command path state machine receives a 48-bit or 136-bit response and sends it to the BIU. If the start bit of the card response is not received within the number of clock cycles (as set up in the t_{mount} register), the r_{to} bit and command done (CD) bit are set to 1 in the r_{intsts} register, to signal to the BIU. If the response-expected bit is set to 0, the command path sends out a command and signals a response done to the BIU, which causes the cmd bit to be set to 1 in the r_{intsts} register.[†]
3. **response_length**—If this bit is set to 1, a 136-bit long response is received; if it is set to 0, a 48-bit short response is received.[†]
4. **check_response_crc**—If this bit is set to 1, the command path compares CRC-7 received in the response with the internally-generated CRC-7. If the two do not match, the response CRC error is signaled to the BIU, that is, the r_{crc} bit is set to 1 in the r_{intsts} register.[†]

Send Response to BIU

If the response_expected bit is set to 1 in the cmd register, the received response is sent to the BIU. Response register 0 (resp0) is updated for a short response, and the response register 3 (resp3), response register 2 (resp2), response register 1 (resp1), and resp0 registers are updated on a long response, after which the cmd bit is set to 1 in the r_{intsts} register. If the response is for an AUTO_STOP command sent by the CIU, the response is written to the resp1 register, after which the auto command done bit (acd) is set to 1 in the r_{intsts} register.[†]

The command path verifies the contents of the card response.

Table 14-11: Card Response Fields[†]

Field	Contents
Response transmission bit	0
Command index	Command index of the sent command
End bit	1

The command index is not checked for a 136-bit response or if the check_response_crc bit in the cmd register is set to 0. For a 136-bit response and reserved CRC 48-bit responses, the command index is reserved, that is, 0b111111.[†]

Related Information

[SD Association](#)

For more information about response values, refer to Physical Layer Simplified Specification, Version 3.01 as described on the SD Association website.

Driving P-bit to the CMD Pin

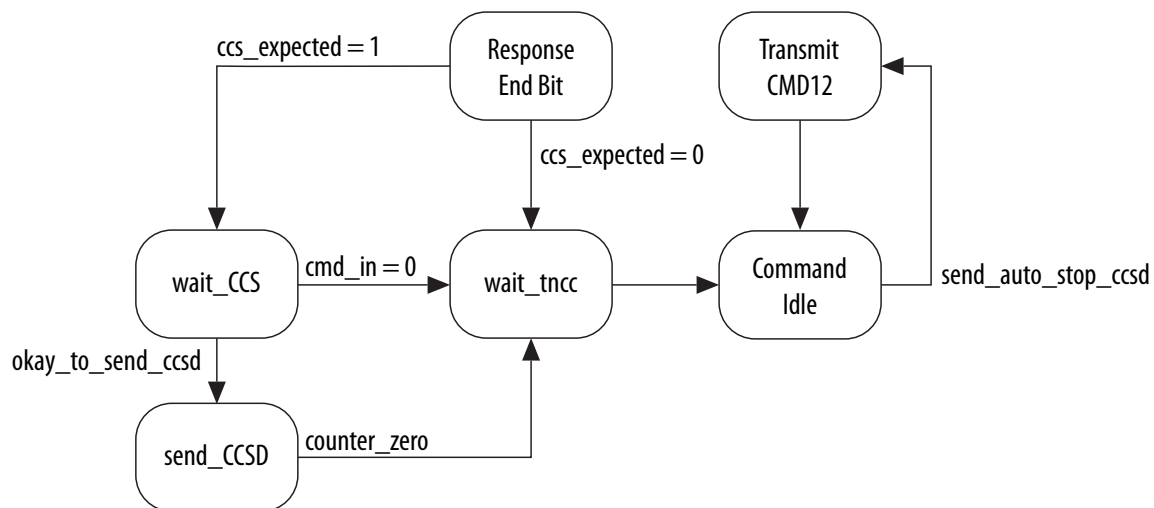
The command path drives a one-cycle pull-up bit (P-bit) to 1 on the CMD pin between two commands if a response is not expected. If a response is expected, the P-bit is driven after the response is received and before the start of the next command. While accessing a CE-ATA card device, for commands that expect a CCS, the P-bit is driven after the response only if the interrupts are disabled in the CE-ATA card (the nIEN

bit is set to 1 in the ATA control register), that is, the CCS expected bit (`ccs_expected`) in the `cmd` register is set to 0. If the command expects the CCS, the P-bit is driven only after receiving the CCS.[†]

Polling the CCS

CE-ATA card devices generate the CCS to notify the host controller of the normal ATA command completion or ATA command termination. After receiving the response from the card, the command path state machine performs the functions illustrated in the following figure according to `cmd` register bit values.[†]

Figure 14-7: CE-ATA Command Path State Machine[†]



The above figure illustrates:

- Response end bit state—The state machine receives the end bit of the response from the card device. If the `ccs_expected` bit of the `cmd` register is set to 1, the state machine enters the wait CCS state.[†]
- Wait CCS—The state machine waits for the CCS from the CE-ATA card device. While waiting for the CCS, the following events can happen:[†]
 1. Software sets the send CCSD bit (`send_ccsd`) in the `ctrl` register, indicating not to wait for CCS and to send the CCSD pattern on the command line.[†]
 2. Receive the CCS on the CMD line.[†]
- Send CCSD command—Sends the CCSD pattern (0b00001) on the CMD line.[†]

CCS Detection and Interrupt to Host Processor

If the `ccs_expected` bit in the `cmd` register is set to 1, the CCS from the CE-ATA card device is indicated by setting the data transfer over bit (`dto`) in the `rintsts` register. The controller generates a DTO interrupt if this interrupt is not masked.[†]

For the `RW_MULTIPLE_BLOCK` commands, if the CE-ATA card device interrupts are disabled (the `nIEN` bit is set to 1 in the ATA control register)—that is, the `ccs_expected` bit is set to 0 in the `cmd` register—there are no CCSs from the card. When the data transfer is over—that is, when the requested number of bytes are transferred—the `dto` bit in the `rintsts` register is set to 1.[†]

CCS Timeout

If the command expects a CCS from the card device (the `ccs_expected` bit is set to 1 in the `cmd` register), the command state machine waits for the CCS and remains in the wait CCS state. If the CE-ATA card fails to send out the CCS, the host software must implement a timeout mechanism to free the command and data path. The controller does not implement a hardware timer; it is the responsibility of the host software to maintain a software timer.[†]

In the event of a CCS timeout, the host must issue a CCSD command by setting the `send_ccsd` bit in the `ctrl` register. The controller command path state machine sends the CCSD command to the CE-ATA card device and exits to an idle state. After sending the CCSD command, the host must also send an SD/SDIO STOP_TRANSMISSION command to the CE-ATA card to abort the outstanding ATA command.[†]

Send CCSD Command

If the `send_ccsd` bit in the `ctrl` register is set to 1, the controller sends a CCSD pattern on the CMD line. The host can send the CCSD command while waiting for the CCS or after a CCS timeout happens.[†]

After sending the CCSD pattern, the controller sets the `cmd` bit in the `rintsts` register and also generates an interrupt to the host if the Command Done interrupt is not masked.[†]

Note: Within the CIU block, if the `send_ccsd` bit in the `ctrl` register is set to 1 on the same clock cycle as CCS is sampled, the CIU block does not send a CCSD pattern on the CMD line. In this case, the `dto` and `cmd` bits in the `rintsts` register are set to 1.[†]

Note: Due to asynchronous boundaries, the CCS might have already happened and the `send_ccsd` bit is set to 1. In this case, the CCSD command does not go to the CE-ATA card device and the `send_ccsd` bit is not set to 0. The host must reset the `send_ccsd` bit to 0 before the next command is issued.[†]

If the send auto stop CCSD (`send_auto_stop_ccsd`) bit in the `ctrl` register is set to 1, the controller sends an internally generated STOP_TRANSMISSION command (CMD12) after sending the CCSD pattern. The controller sets the `acd` bit in the `rintsts` register.[†]

I/O transmission delay (N_{ACIO} Timeout)

The host software maintains the timeout mechanism for handling the I/O transmission delay (N_{ACIO} cycles) time-outs while reading from the CE-ATA card device. The controller neither maintains any timeout mechanism nor indicates that N_{ACIO} cycles are elapsed while waiting for the start bit of a data token. The I/O transmission delay is applicable for read transfers using the `RW_REG` and `RW_BLK` commands; the `RW_REG` and `RW_BLK` commands used in this document refer to the `RW_MULTIPLE_REGISTER` and `RW_MULTIPLE_BLOCK` MMC commands defined by the CE-ATA specification.[†]

Note: After the N_{ACIO} timeout, the application must abort the command by sending the CCSD and STOP commands, or the STOP command. The Data Read Timeout (DRT0) interrupt might be set to 1 while a STOP_TRANSMISSION command is transmitted out of the controller, in which case the data read timeout boot data start bit (`bds`) and the `dto` bit in the `rintsts` register are set to 1.[†]

Data Path

The data path block reads the data FIFO buffer and transmits data on the card bus during a write data transfer, or receives data and writes it to the FIFO buffer during a read data transfer. The data path loads new data parameters—data expected, read/write data transfer, stream/block transfer, block size, byte count, card type, timeout registers—whenever a data transfer command is not in progress. If the data

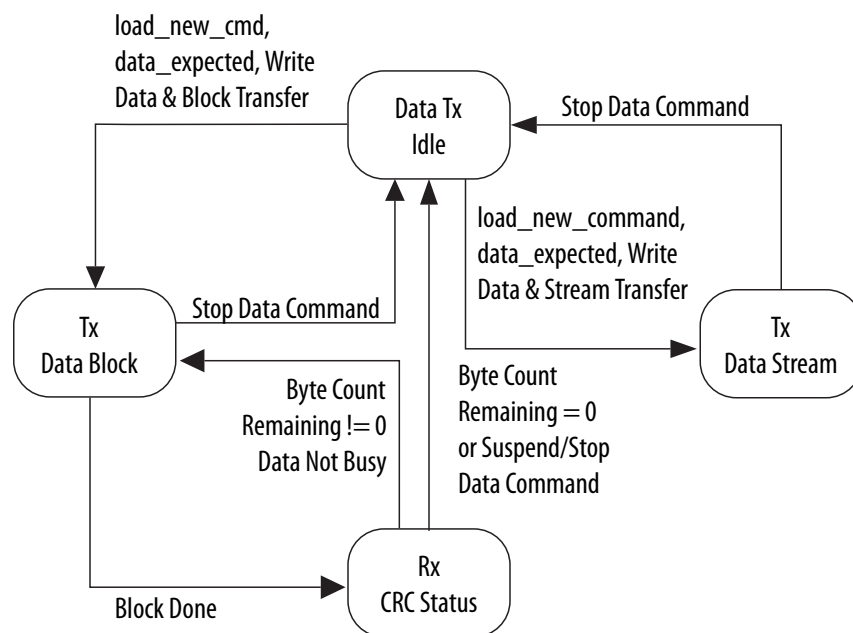
transfer expected bit (`data_expected`) in the `cmd` register is set to 1, the new command is a data transfer command and the data path starts one of the following actions:[†]

- Transmits data if the read/write bit = 1[†]
- Receives data if read/write bit = 0[†]

Data Transmit

The data transmit state machine starts data transmission two clock cycles after a response for the data write command is received. This occurs even if the command path detects a response error or response CRC error. If a response is not received from the card because of a response timeout, data is not transmitted. Depending upon the value of the transfer mode bit (`transfer_mode`) in the `cmd` register, the data transmit state machine puts data on the card data bus in a stream or in blocks.[†]

Figure 14-8: Data Transmit State Machine[†]



Stream Data Transmit

If the `transfer_mode` bit in the `cmd` register is set to 1, the transfer is a stream-write data transfer. The data path reads data from the FIFO buffer from the BIU and transmits in a stream to the card data bus. If the FIFO buffer becomes empty, the card clock is stopped and restarted once data is available in the FIFO buffer.[†]

If the `bytcnt` register is reset to 0, the transfer is an open-ended stream-write data transfer. During this data transfer, the data path continuously transmits data in a stream until the host software issues an SD/SDIO STOP command. A stream data transfer is terminated when the end bit of the STOP command and end bit of the data match over two clock cycles.[†]

If the `bytcnt` register is written with a nonzero value and the `send_auto_stop` bit in the `cmd` register is set to 1, the STOP command is internally generated and loaded in the command path when the end bit of the STOP command occurs after the last byte of the stream write transfer matches. This data transfer can also terminate if the host issues a STOP command before all the data bytes are transferred to the card bus.[†]

Single Block Data

If the `transfer_mode` bit in the `cmd` register is set to 0 and the `bytcnt` register value is equal to the value of the `block_size` register, a single-block write-data transfer occurs. The data transmit state machine sends data in a single block, where the number of bytes equals the block size, including the internally-generated 16-term CRC (CRC-16).[†]

If the `ctype` register is set for a 1-bit, 4-bit, or 8-bit data transfer, the data is transmitted on 1, 4, or 8 data lines, respectively, and CRC-16 is separately generated and transmitted for 1, 4, or 8 data lines, respectively.[†]

After a single data block is transmitted, the data transmit state machine receives the CRC status from the card and signals a data transfer to the BIU. This happens when the `dto` bit in the `rintsts` register is set to 1.[†]

If a negative CRC status is received from the card, the data path signals a data CRC error to the BIU by setting the `dcrc` bit in the `rintsts` register.[†]

Additionally, if the start bit of the CRC status is not received by two clock cycles after the end of the data block, a CRC status start-bit error (SBE) is signaled to the BIU by setting the `sbe` bit in the `rintsts` register.[†]

Multiple Block Data

A multiple-block write-data transfer occurs if the `transfer_mode` bit in the `cmd` register is set to 0 and the value in the `bytcnt` register is not equal to the value of the `block_size` register. The data transmit state machine sends data in blocks, where the number of bytes in a block equals the block size, including the internally-generated CRC-16 value.[†]

If the `ctype` register is set to 1-bit, 4-bit, or 8-bit data transfer, the data is transmitted on 1-, 4-, or 8-data lines, respectively, and CRC-16 is separately generated and transmitted on 1-, 4-, or 8-data lines, respectively.[†]

After one data block is transmitted, the data transmit state machine receives the CRC status from the card. If the remaining byte count becomes 0, the data path signals to the BIU that the data transfer is done. This happens when the `dto` bit in the `rintsts` register is set to 1.[†]

If the remaining data bytes are greater than zero, the data path state machine starts to transmit another data block.[†]

If a negative CRC status is received from the card, the data path signals a data CRC error to the BIU by setting the `dcrc` bit in the `rintsts` register, and continues further data transmission until all the bytes are transmitted.[†]

If the CRC status start bit is not received by two clock cycles after the end of a data block, a CRC status SBE is signaled to the BIU by setting the `ebe` bit in the `rintsts` register and further data transfer is terminated.[†]

If the `send_auto_stop` bit is set to 1 in the `cmd` register, the SD/SDIO STOP command is internally generated during the transfer of the last data block, where no extra bytes are transferred to the card. The end bit of the STOP command might not exactly match the end bit of the CRC status in the last data block.[†]

If the block size is less than 4, 16, or 32 for card data widths of 1 bit, 4 bits, or 8 bits, respectively, the data transmit state machine terminates the data transfer when all the data is transferred, at which time the internally-generated STOP command is loaded in the command path.[†]

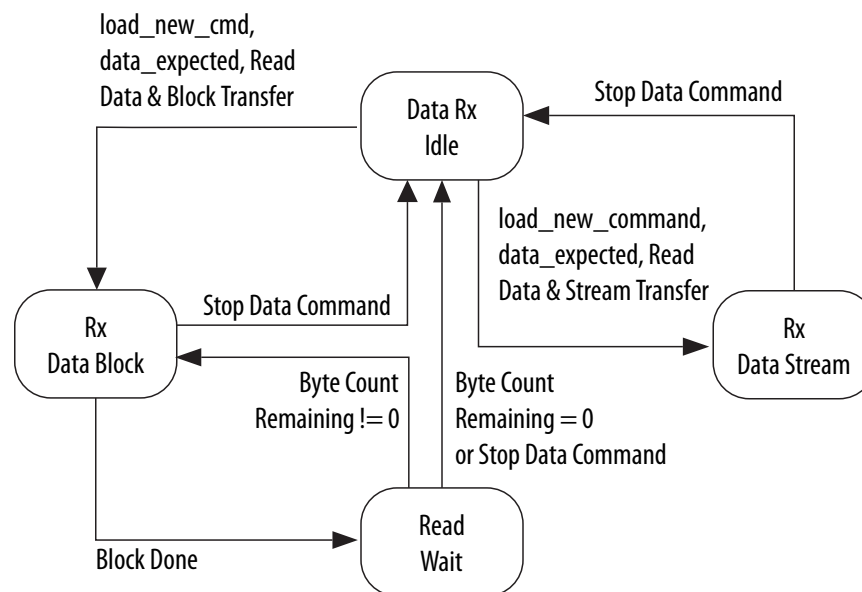
If the `bytcnt` is zero (the block size must be greater than zero) the transfer is an open-ended block transfer. The data transmit state machine for this type of data transfer continues the block-write data transfer until the host software issues an SD/SDIO STOP or STOP_TRANSMISSION (CMD12) command.[†]

Data Receive

The data-receive state machine receives data two clock cycles after the end bit of a data read command, even if the command path detects a response error or response CRC error. If a response is not received from the card because a response timeout occurs, the BIU does not receive a signal that the data transfer is complete. This happens if the command sent by the controller is an illegal operation for the card, which keeps the card from starting a read data transfer.[†]

If data is not received before the data timeout, the data path signals a data timeout to the BIU and an end to the data transfer done. Based on the value of the `transfer_mode` bit in the `cmd` register, the data-receive state machine gets data from the card data bus in a stream or block(s).[†]

Figure 14-9: Data Receive State Machine[†]



Stream Data Read

A stream-read data transfer occurs if the `transfer_mode` bit in the `cmd` register is set to 1, at which time the data path receives data from the card and writes it to the FIFO buffer. If the FIFO buffer becomes full, the card clock stops and restarts once the FIFO buffer is no longer full.[†]

An open-ended stream-read data transfer occurs if the `bytcnt` register is set to 0. During this type of data transfer, the data path continuously receives data in a stream until the host software issues an SD/SDIO STOP command. A stream data transfer terminates two clock cycles after the end bit of the STOP command.[†]

If the `bytcnt` register contains a nonzero value and the `send_auto_stop` bit in the `cmd` register is set to 1, a STOP command is internally generated and loaded into the command path, where the end bit of the STOP command occurs after the last byte of the stream data transfer is received. This data transfer can

terminate if the host issues an SD/SDIO STOP or STOP_TRANSMISSION (CMD12) command before all the data bytes are received from the card.[†]

Single-block Data Read

If the `ctype` register is set to a 1-bit, 4-bit, or 8-bit data transfer, data is received from 1, 4, or 8 data lines, respectively, and CRC-16 is separately generated and checked for 1, 4, or 8 data lines, respectively. If there is a CRC-16 mismatch, the data path signals a data CRC error to the BIU. If the received end bit is not 1, the BIU receives an End-bit Error (EBE).[†]

Multiple-block Data Read

If the `transfer_mode` bit in the `cmd` register is clear and the value of the `bytcnt` register is not equal to the value of the `block_size` register, the transfer is a multiple-block read-data transfer. The data-receive state machine receives data in blocks, where the number of bytes in a block is equal to the block size, including the internally-generated CRC-16.[†]

If the `ctype` register is set to a 1-bit, 4-bit, or 8-bit data transfer, data is received from 1, 4, or 8 data lines, respectively, and CRC-16 is separately generated and checked for 1, 4, or 8 data lines, respectively. After a data block is received, if the remaining byte count becomes zero, the data path signals a data transfer to the BIU.[†]

If the remaining data bytes are greater than zero, the data path state machine causes another data block to be received. If CRC-16 of a received data block does not match the internally-generated CRC-16, a data CRC error to the BIU and data reception continue further data transmission until all bytes are transmitted. Additionally, if the end of a received data block is not 1, data on the data path signals terminate the bit error to the CIU and the data-receive state machine terminates data reception, waits for data timeout, and signals to the BIU that the data transfer is complete.[†]

If the `send_auto_stop` bit in the `cmd` register is set to 1, the SD/SDIO STOP command is internally generated when the last data block is transferred, where no extra bytes are transferred from the card. The end bit of the STOP command might not exactly match the end bit of the last data block.[†]

If the requested block size for data transfers to cards is less than 4, 16, or 32 bytes for 1-bit, 4-bit, or 8-bit data transfer modes, respectively, the data-transmit state machine terminates the data transfer when all data is transferred, at which point the internally-generated STOP command is loaded in the command path. Data received from the card after that are then ignored by the data path.[†]

If the `bytcnt` register is 0 (the block size must be greater than zero), the transfer is an open-ended block transfer. For this type of data transfer, the data-receive state machine continues the block-read data transfer until the host software issues an SD/SDIO STOP or STOP_TRANSMISSION (CMD12) command.[†]

Auto-Stop

The controller internally generates an SD/SDIO STOP command and is loaded in the command path when the `send_auto_stop` bit in the `cmd` register is set to 1. The AUTO_STOP command helps to send an exact number of data bytes using a stream read or write for the MMC, and a multiple-block read or write for SD memory transfer for SD cards. The software must set the `send_auto_stop` bit according to the following details: [†]

The following list describes conditions for the AUTO_STOP command:[†]

- Stream-read for MMC with byte count greater than zero—The controller generates an internal STOP command and loads it into the command path so that the end bit of the STOP command is sent when the last byte of data is read from the card and no extra data byte is received. If the byte count is less than six (48 bits), a few extra data bytes are received from the card before the end bit of the STOP command is sent.[†]
- Stream-write for MMC with byte count greater than zero—The controller generates an internal STOP command and loads it into the command path so that the end bit of the STOP command is sent when the last byte of data is transmitted on the card bus and no extra data byte is transmitted. If the byte count is less than six (48 bits), the data path transmits the data last to meet these condition.[†]
- Multiple-block read memory for SD card with byte count greater than zero—If the block size is less than four (single-bit data bus), 16 (4-bit data bus), or 32 (8-bit data bus), the AUTO_STOP command is loaded in the command path after all the bytes are read. Otherwise, the STOP command is loaded in the command path so that the end bit of the STOP command is sent after the last data block is received.[†]
- Multiple-block write memory for SD card with byte count greater than zero—If the block size is less than three (single-bit data bus), 12 (4-bit data bus), or 24 (8-bit data bus), the AUTO_STOP command is loaded in the command path after all data blocks are transmitted. Otherwise, the STOP command is loaded in the command path so that the end bit of the STOP command is sent after the end bit of the CRC status is received.[†]
- Precaution for host software during auto-stop—When an AUTO_STOP command is issued, the host software must not issue a new command to the controller until the AUTO_STOP command is sent by the controller and the data transfer is complete. If the host issues a new command during a data transfer with the AUTO_STOP command in progress, an AUTO_STOP command might be sent after the new command is sent and its response is received. This can delay sending the STOP command, which transfers extra data bytes. For a stream write, extra data bytes are erroneous data that can corrupt the card data. If the host wants to terminate the data transfer before the data transfer is complete, it can issue an SD/SDIO STOP or STOP_TRANSMISSION (CMD12) command, in which case the controller does not generate an AUTO_STOP command.[†]

Auto-Stop Generation for MMC Cards

Table 14-12: Auto-Stop Generation for MMC Cards[†]

Transfer Type	Byte Count	send_auto_stop bit set	Comments
Stream read	0	No	Open-ended stream
Stream read	>0	Yes	Auto-stop after all bytes transfer
Stream write	0	No	Open-ended stream
Stream write	>0	Yes	Auto-stop after all bytes transfer
Single-block read	>0	No	Byte count = 0 is illegal
Single-block write	>0	No	Byte count = 0 is illegal
Multiple-block read	0	No	Open-ended multiple block
Multiple-block read	>0	Yes ^{(44)†}	Pre-defined multiple block

⁽⁴⁴⁾ The condition under which the transfer mode is set to block transfer and byte_count is equal to block size is treated as a single-block data transfer command for both MMC and SD cards. If byte_count = n*block_size (n = 2, 3, ...), the condition is treated as a predefined multiple-block data transfer command. In the case of

Transfer Type	Byte Count	send_auto_stop bit set	Comments
Multiple-block write	0	No	Open-ended multiple block
Multiple-block write	>0	Yes ⁽⁴⁴⁾ †	Pre-defined multiple block

Auto-Stop Generation for SD Cards

Table 14-13: Auto-Stop Generation for SD Cards[†]

Transfer Type	Byte Count	send_auto_stop bit set	Comments
Single-block read	>0	No	Byte count = 0 is illegal
Single-block write	>0	No	Byte count = 0 illegal
Multiple-block read	0	No	Open-ended multiple block
Multiple-block read	>0	Yes	Auto-stop after all bytes transfer
Multiple-block write	0	No	Open-ended multiple block
Multiple-block write	>0	Yes	Auto-stop after all bytes transfer

Auto-Stop Generation for SDIO Cards

Table 14-14: Auto-Stop Generation for SDIO Cards[†]

Transfer Type	Byte Count	send_auto_stop bit set	Comments
Single-block read	>0	No	Byte count = 0 is illegal
Single-block write	>0	No	Byte count = 0 illegal
Multiple-block read	0	No	Open-ended multiple block
Multiple-block read	>0	No	Pre-defined multiple block
Multiple-block write	0	No	Open-ended multiple block
Multiple-block write	>0	No	Pre-defined multiple block

Non-Data Transfer Commands that Use Data Path

Some SD/SDIO non-data transfer commands (commands other than read and write commands) also use the data path.

an MMC card, the host software can perform a predefined data transfer in two ways: 1) Issue the CMD23 command before issuing CMD18/CMD25 commands to the card – in this case, issue CMD18/CMD25 commands without setting the send_auto_stop bit. 2) Issue CMD18/CMD25 commands without issuing CMD23 command to the card, with the send_auto_stop bit set. In this case, the multiple-block data transfer is terminated by an internally-generated auto-stop command after the programmed byte count.[†]

Table 14-15: Non-Data Transfer Commands and Requirements[†]

	PROGRAM_CSD (CMD27)	SEND_WRITE_PROT (CMD30)	LOCK_UNLOCK (CMD42)	SD_STATUS (ACMD13)	SEND_NUM_WR_BLOCKS (ACMD22)	SEND_SCR (ACMD51)
Command register programming [†]						
Cmd_index	0x1B=27	0x1E=30	0x2A=42	0x0D=13	0x16=22	0x33=51
Response_expect [†]	1	1	1	1	1	1
Response_length [†]	0	0	0	0	0	0
Check_response_crc [†]	1	1	1	1	1	1
Data_expected [†]	1	1	1	1	1	1
Read/write [†]	1	0	1	0	0	0
Transfer_mode [†]	0	0	0	0	0	0
Send_auto_stop [†]	0	0	0	0	0	0
Wait_prevdata_complete [†]	0	0	0	0	0	0
Stop_abort_cmd [†]	0	0	0	0	0	0

Table 14-16: Non-Data Transfer Commands and Requirements (Cont.)[†]

	PROGRAM_CSD (CMD27)	SEND_WRITE_PROT (CMD30)	LOCK_UNLOCK (CMD42)	SD_STATUS (ACMD13)	SEND_NUM_WR_BLOCKS (ACMD22)	SEND_SCR (ACMD51)
Command Argument register programming [†]						
	Stuff bits	32-bit write protect data address	Stuff bits	Stuff bits	Stuff bits	Stuff bits

Table 14-17: Non-Data Transfer Commands and Requirements[†]

PROGRAM_CSD (CMD27)	SEND_WRITE_PROT (CMD30)	LOCK_UNLOCK (CMD42)	SD_STATUS (ACMD13)	SEND_NUM_WR_BLOCKS (ACMD22)	SEND_SCR (ACMD51)
Block Size register programming [†]					
16	4	Num_bytes ⁽⁴⁵⁾	64	4	8

Table 14-18: Non-Data Transfer Commands and Requirements[†]

PROGRAM_CSD (CMD27)	SEND_WRITE_PROT (CMD30)	LOCK_UNLOCK (CMD42)	SD_STATUS (ACMD13)	SEND_NUM_WR_BLOCKS (ACMD22)	SEND_SCR (ACMD51)
Byte Count register programming [†]					
16	4	Num_bytes ⁽⁴⁶⁾	64	4	8

Related Information

- **SD Association**
For more information, the SD specification can be purchased from this organization.
- **JEDEC Global Standards of the Microelectronics Industry**
For more information, the MMC specification can be purchased from this organization.

Clock Control Block

The clock control block provides different clock frequencies required for SD/MMC/CE-ATA cards. The clock control block has one clock divider, which is used to generate different card clock frequencies.[†]

The clock frequency of a card depends on the following clock ctrl register settings:[†]

⁽⁴⁵⁾ Num_bytes = Number of bytes specified as per the lock card data structure. Refer to the SD specification and the MMC specification.[†]

⁽⁴⁶⁾ Num_bytes = Number of bytes specified as per the lock card data structure. Refer to the SD specification and the MMC specification.[†]

- **clkdiv register**—Internal clock dividers are used to generate different clock frequencies required for the cards. The division factor for the clock divider can be set by writing to the **clkdiv** register. The clock divider is an 8-bit value that provides a clock division factor from 1 to 510; a value of 0 represents a clock-divider bypass, a value of 1 represents a divide by 2, a value of 2 represents a divide by 4, and so on.[†]
- **clksrc register**—Set this register to 0 as clock is divided by clock divider 0.[†]
- **clkena register**—The **ccclk_out** card output clock can be enabled or disabled under the following conditions:[†]
 - **ccclk_out** is enabled when the **ccclk_enable** bit in the **clkena** register is set to 1 and disabled when set to 0.[†]
 - Low-power mode can be enabled by setting the **ccclk_low_power** bit of the **clkena** register to 1. If low-power mode is enabled to save card power, the **ccclk_out** signal is disabled when the card is idle for at least eight card clock cycles. Low-power mode is enabled when a new command is loaded and the command path goes to a non-idle state.[†]

Under the following conditions, the card clock is stopped or disabled:[†]

- Clock can be disabled by writing to the **clkena** register.[†]
- When low-power mode is selected and the card is idle for at least eight clock cycles.[†]
- FIFO buffer is full, data path cannot accept more data from the card, and data transfer is incomplete—to avoid FIFO buffer overflow.[†]
- FIFO buffer is empty, data path cannot transmit more data to the card, and data transfer is incomplete—to avoid FIFO buffer underflow.[†]

Note: The card clock must be disabled through the **clkena** register before the host software changes the values of the **clkdiv** and **clksrc** registers.[†]

Error Detection

Errors can occur during card operations within the CIU in the following situations.

Response[†]

- Response timeout—did not receive the response expected with response start bit within the specified number of clock cycles in the timeout register.[†]
- Response CRC error—response is expected and check response CRC requested; response CRC-7 does not match with the internally-generated CRC-7.[†]
- Response error—response transmission bit is not 0, command index does not match with the command index of the send command, or response end bit is not 1.[†]

Data Transmit[†]

- No CRC status—during a write data transfer, if the CRC status start bit is not received for two clock cycles after the end bit of the data block is sent out, the data path performs the following actions:[†]
 - Signals no CRC status error to the BIU[†]
 - Terminates further data transfer[†]
 - Signals data transfer done to the BIU[†]
- Negative CRC—if the CRC status received after the write data block is negative (that is, not 0b010), the data path signals a data CRC error to the BIU and continues with the data transfer.[†]
- Data starvation due to empty FIFO buffer—if the FIFO buffer becomes empty during a write data transmission, or if the card clock stopped and the FIFO buffer remains empty for a data-timeout number of clock cycles, the data path signals a data-starvation error to the BIU and the data path continues to wait for data in the FIFO buffer.[†]

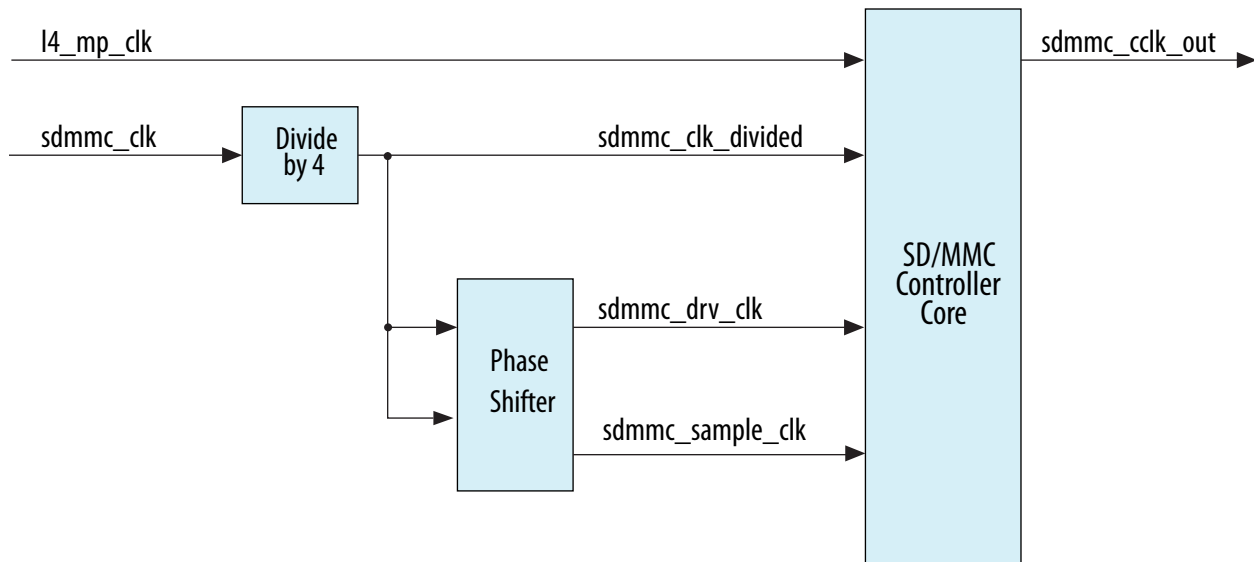
Data Receive

- Data timeout—during a read-data transfer, if the data start bit is not received before the number of clock cycles specified in the timeout register, the data path does the following action: [†]
 - Signals a data-timeout error to the BIU[†]
 - Terminates further data transfer[†]
 - Signals data transfer done to BIU[†]
- Data SBE—during a 4-bit or 8-bit read-data transfer, if the all-bit data line does not have a start bit, the data path signals a data SBE to the BIU and waits for a data timeout, after which it signals that the data transfer is done.[†]
- Data CRC error—during a read-data-block transfer, if the CRC-16 received does not match with the internally generated CRC-16, the data path signals a data CRC error to the BIU and continues with the data transfer.[†]
- Data EBE—during a read-data transfer, if the end bit of the received data is not 1, the data path signals an EBE to the BIU, terminates further data transfer, and signals to the BIU that the data transfer is done.[†]
- Data starvation due to FIFO buffer full—during a read data transmission and when the FIFO buffer becomes full, the card clock stops. If the FIFO buffer remains full for a data-timeout number of clock cycles, the data path signals a data starvation error to the BIU, by setting the data starvation host timeout bit (hto) in rintsts register to 1, and the data path continues to wait for the FIFO buffer to empty.[†]

Clocks

Table 14-19: SD/MMC Controller Clocks

Clock Name	Direction	Description
l4_mp_clk	In	Clock for SD/MMC controller BIU
sdmmc_clk	In	Clock for SD/MMC controller
sdmmc_cclk_out	Out	Generated output clock for card
sdmmc_clk_divided	Internal	Divide-by-four clock of sdmmc_clk
sdmmc_sample_clk	Internal	Phase-shifted clock of sdmmc_clk_divided used to sample the command and data from the card
sdmmc_drv_clk	Internal	Phase-shifted clock of sdmmc_clk_divided for controller to drive command and data to the card to meet hold time requirements

Figure 14-10: SD/MMC Controller Clock Connections

The `sdmmc_clk` clock from the clock manager is divided by four and becomes the `sdmmc_clk_divided` clock before passing to the phase shifters and the SD/MMC controller CIU. The phase shifters are used to generate the `sdmmc_drv_clk` and `sdmmc_sample_clk` clocks. These phase shifters provide up to eight phases shift which include 0, 45, 90, 135, 180, 225, 270, and 315 degrees. The `sdmmc_sample_clk` clock can be driven by the output from the phase shifter.

Note: The selections of phase shift degree and `sdmmc_sample_clk` source are done in the system manager. For information about setting the phase shift and selecting the source of the `sdmmc_sample_clk` clock, refer to the "Clock Setup" section within this document.

The controller generates the `sdmmc_cclk_out` clock, which is driven to the card. For more information about the generation of the `sdmmc_cclk_out` clock, refer to the "Clock Control Block" section within this document.

Related Information

- **Clock Setup** on page 14-46
Refer to this section for information about setting the phase shift.
- **Clock Control Block** on page 14-33
Refer to this section for information about the generation of the `sdmmc_cclk_out` clock.

Resets

The SD/MMC controller has one reset signal. The reset manager drives this signal to the SD/MMC controller on a cold or warm reset.

Related Information

Reset Manager on page 4-1

Taking the SD/MMC Controller Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

Related Information

[Module Reset Signals](#) on page 4-5

Voltage Switching

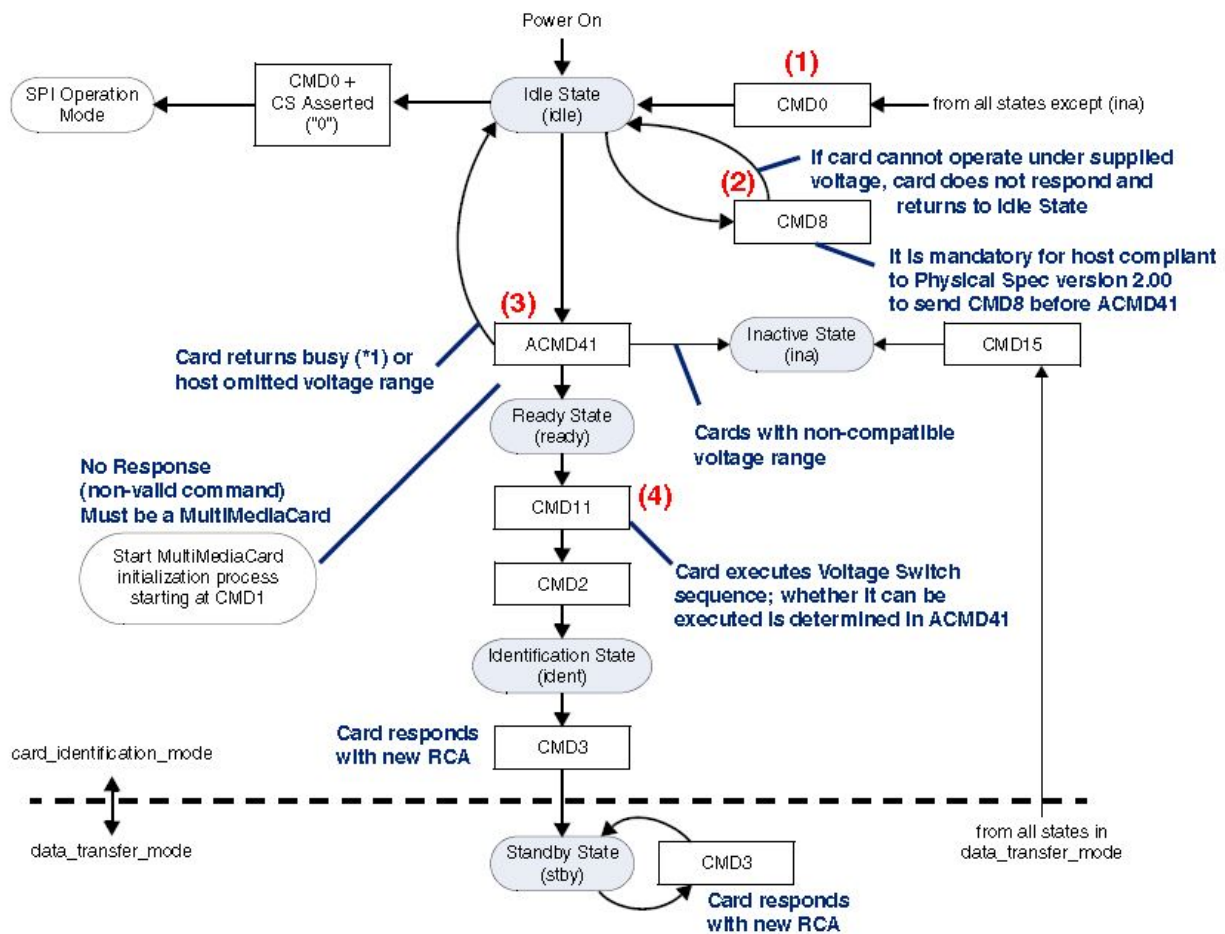
This section describes the general steps to switch voltage level.

The SD/MMC cards support various operating voltages, for example 1.8V and 3.3V. If you have a card which is at 1.8V and you eject it and replace it with another card, which is 3.3V, then voltage switching is required.

In order to have the right voltage level to power the card, separate devices on the board are required: voltage translation transceiver and power regulator/supply. When the software is aware that voltage switching is needed, it notifies the power regulator that it needs to supply another voltage level to the card (switching between 1.8V and 3.3V).

Many SD cards have an option to signal at 1.8 or 3.3 V, however the initial power-up voltage requirement is 3.3V. To support these different voltage requirements, external transceivers are needed.

The general steps to switch the voltage level requires you to use a SD/MMC voltage-translation transceiver in between the HPS and the SD/MMC card.

Figure 14-11: Voltage Switching Command Flow Diagram[†]

(*1) **Note:** Card returns busy when:[†]

- Card executes internal initialization process[†]
- Card is High or Extended capacity SD Memory Card and host does not support High[†]

The following outlines the steps for the voltage switch programming sequence.[†]

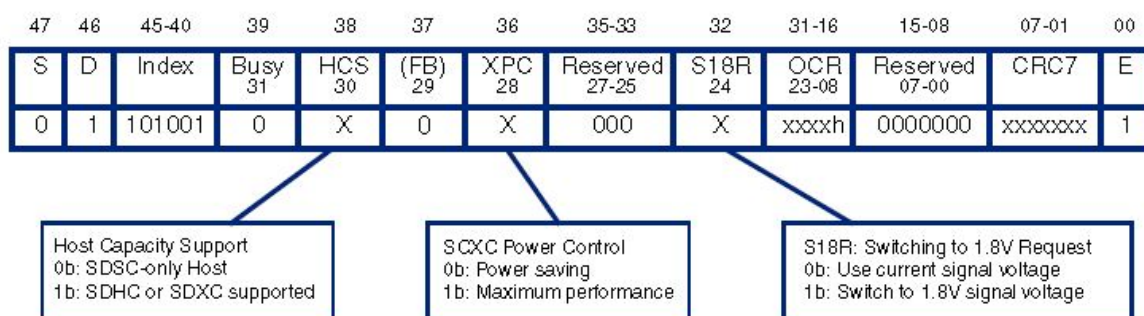
1. Software Driver starts CMD0, which selects the bus mode as SD.[†]
2. After the bus is in SD card mode, CMD8 is started in order to verify if the card is compatible with the SD Memory Card Specification, Version 2.00.[†]

CMD8 determines if the card is capable of working within the host supply voltage specified in the VHS (19:16) field of the CMD; the card supports the current host voltage if a response to CMD8 is received.[†]

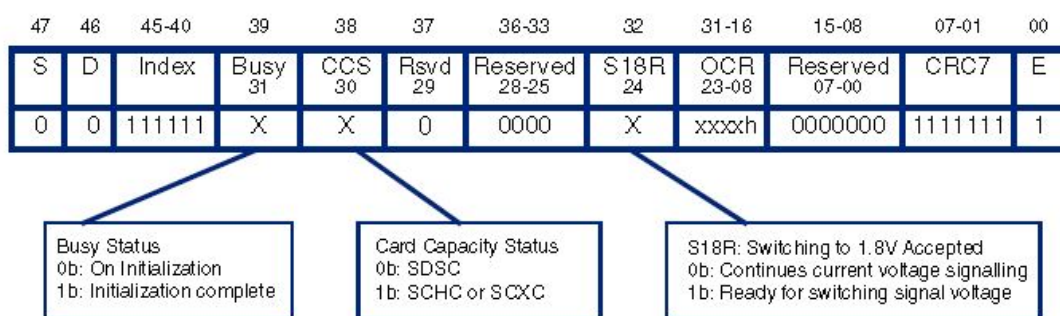
3. ACMD 41 is started.[†]

The response to this command informs the software if the card supports voltage switching; bits 38, 36, and 32 are checked by the card argument of ACMD41.[†]

Figure 14-12: ACMD41 Argument



- a. Bit 30 informs the card if host supports SDHC/SDXC or not; this bit should be set to 1'b1.[†]
- b. Bit 28 can be either 1 or 0.[†]
- c. Bit 24 should be set to 1'b1, indicating that the host is capable of voltage switching.[†]

Figure 14-13: ACMD41 Response (R3)[†]

- d. Bit 30 – If set to 1'b1, card supports SDHC/SDXC; if set to 1'b0, card supports only SDSC.[†]
 - e. Bit 24 – If set to 1'b1, card supports voltage switching and is ready for the switch.[†]
 - f. Bit 31 – If set to 1'b1, initialization is over; if set to 1'b0, means initialization in process[†]
4. If the card supports voltage switching, then the software must perform the steps discussed for either the “Voltage Switch Normal Scenario” or the “Voltage Switch Error Scenario”, located in the *Synopsys DesignWare Cores Mobile Storage Host Databook*.

Related Information

[Synopsys DesignWare Cores Mobile Storage Host Databook](#)

For more information about Voltage Switching

SD/MMC Controller Programming Model

Software and Hardware Restrictions[†]

Only one data transfer command should be issued at one time. For CE-ATA devices, if CE-ATA device interrupts are enabled (nIEN=0), only one RW_MULTIPLE_BLOCK command (RW_BLK) should be

issued; no other commands (including a new RW_BLK) should be issued before the Data Transfer Over status is set for the outstanding RW_BLK.[†]

Before issuing a new data transfer command, the software should ensure that the card is not busy due to any previous data transfer command. Before changing the card clock frequency, the software must ensure that there are no data or command transfers in progress.[†]

If the card is enumerated in SDR12 or SDR25 mode, the application must program the use_hold_reg bit[29] in the CMD register to 1'b1.[†]

This programming should be done for all data transfer commands and non-data commands that are sent to the card. When the use_hold_reg bit is programmed to 1'b0, the SD/MMC controller bypasses the Hold Registers in the transmit path. The value of this bit should not be changed when a Command or Data Transfer is in progress.[†]

For more information on using the use_hold_reg and the implementation requirements for meeting the card input hold time, refer to the latest version of the Synopsys DesignWare Cores Mobile Storage Host Databook.

Avoiding Glitches in the Card Clock Outputs[†]

To avoid glitches in the card clock outputs (sdmmc_clk_out), the software should use the following steps when changing the card clock frequency:[†]

1. Before disabling the clocks, ensure that the card is not busy due to any previous data command. To determine this, check for 0 in bit 9 of the STATUS register.[†]
2. Update the Clock Enable register to disable all clocks. To ensure completion of any previous command before this update, send a command to the CIU to update the clock registers by setting:[†]
 - start_cmd bit[†]
 - "update clock registers only" bits[†]
 - "wait_previous data complete"[†]

Note: Wait for the CIU to take the command by polling for 0 on the start_cmd bit.[†]

3. Set the start_cmd bit to update the Clock Divider or Clock Source register, or both, Send a command to the CIU to update the clock registers. Wait for the CIU to take the command.
4. Set start_cmd to update the Clock Enable register in order to enable the required clocks and send a command to the CIU to update the clock registers; wait for the CIU to take the command.[†]

Reading from a Card in Non-DMA Mode[†]

When a card is read in non-DMA mode, the Data Transfer Over (RINTSTS[3]) interrupt occurs as soon as the data transfer from the card is over. There still could be some data left in the FIFO, and the RX_WMark interrupt may or may not occur, depending on the remaining bytes in the FIFO. Software should read any remaining bytes upon seeing the Data Transfer Over (DTO) interrupt. While using the external DMA interface for reading from a card, the DTO interrupt occurs only after all the data is flushed to memory by the DMA Interface unit.[†]

Writing to a Card in External DMA Mode[†]

While writing to a card in external DMA mode, if an undefined-length transfer is selected by setting the Byte Count register to 0, the DMA logic may request more data than it sends to the card, since it has no way of knowing at which point the software stops the transfer. The DMA request stops as soon as the DTO is set by the CIU.[†]

Software Issues a controller_Reset Command[†]

If the software issues a `controller_reset` command by setting control register bit[0] to 1, all the CIU state machines are reset; the FIFO is not cleared. The DMA sends all remaining bytes to the host. In addition to a card-reset, if a FIFO reset is also issued, then:[†]

- Any pending DMA transfer on the bus completes correctly[†]
- DMA data read is ignored[†]
- Write data is unknown (x)[†]

Additionally, if `dma_reset` is also issued, any pending DMA transfer is abruptly terminated. When the DW-DMA/Non-DW-DMA is used, the DMA controller channel should also be reset and reprogrammed.[†]

If any of the previous data commands do not properly terminate, then the software should issue the FIFO reset in order to remove any residual data, if any, in the FIFO. After asserting the FIFO reset, you should wait until this bit is cleared.[†]

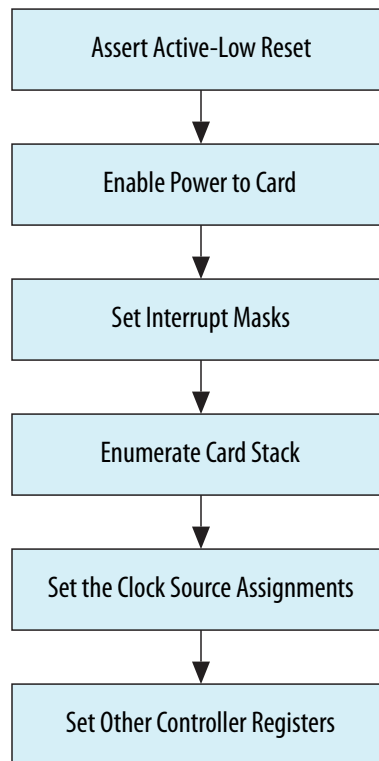
Data-Transfer Requirement Between the FIFO and Host[†]

One data-transfer requirement between the FIFO and host is that the number of transfers should be a multiple of the FIFO data width (`F_DATA_WIDTH`). For example, if `F_DATA_WIDTH` = 32 and you want to write only 15 bytes to an `SD_MMC_CEATA` card (`BYTCNT`), the host should write 16 bytes to the FIFO or program the DMA to do 16-byte transfers, if external DMA mode is enabled. The software can still program the Byte Count register to only 15, at which point only 15 bytes are transferred to the card. Similarly, when 15 bytes are read from a card, the host should still read all 16 bytes from the FIFO.[†]

It is recommended that you not change the FIFO threshold register in the middle of data transfers when DW-DMA/Non-DW-DMA mode is chosen.[†]

Initialization[†]

After the power and clock to the controller are stable, the controller active-low reset is asserted. The reset sequence initializes the registers, FIFO buffer pointers, DMA interface controls, and state machines in the controller.[†]

Figure 14-14: SD/MMC Controller Initialization Sequence[†]

Power-On Reset Sequence

Software must perform the following steps after the power-on-reset:

1. Before enabling power to the card, confirm that the voltage setting to the voltage regulator is correct.[†]
2. Enable power to the card by setting the power enable bit (`power_enable`) in the power enable register (`pwren`) to 1. Wait for the power ramp-up time before proceeding to the next step.[†]
3. Set the interrupt masks by resetting the appropriate bits to 0 in the `intmask` register.[†]
4. Set the `int_enable` bit of the `ctrl` register to 1.[†]

Note: Intel recommends that you write `0xFFFFFFFF` to the `rintsts` register to clear any pending interrupts before setting the `int_enable` bit to 1.[†]

5. Discover the card stack according to the card type. For discovery, you must restrict the clock frequency to 400 kHz in accordance with SD/MMC/CE-ATA standards. For more information, refer to *Enumerated Card Stack*.[†]
6. Set the clock source assignments. Set the card frequency using the `clkdiv` and `clksrc` registers of the controller. For more information, refer to *Clock Setup*.[†]
7. The following common registers and fields can be set during initialization process:[†]

- The response timeout field (`response_timeout`) of the `tmout` register. A typical value is 0×40 .[†]
- The data timeout field (`data_timeout`) of the `tmout` register, highest of the following:[†]
 - $10 * N_{AC}$ [†]
 N_{AC} = card device total access time[†]
 $= 10 * ((TAAC * F_{OP}) + (100 * NSAC))$ [†]
 where:[†]
 $TAAC$ = Time-dependent factor of the data access time[†]
 F_{OP} = The card clock frequency used for the card operation[†]
 $NSAC$ = Worst-case clock rate-dependent factor of the data access time[†]
 - Host FIFO buffer latency[†]
 On read: Time elapsed before host starts reading from a full FIFO buffer[†]
 On write: Time elapsed before host starts writing to an empty FIFO buffer[†]
 - Debounce counter register (`debncn`). A typical debounce value is 25 ms.[†]
 - TX watermark field (`tx_wmark`) of the FIFO threshold watermark register (`fifoth`). Typically, the threshold value is set to 512, which is half the FIFO buffer depth.[†]
 - RX watermark field (`rx_wmark`) of the `fifoth` register. Typically, the threshold value is set to 511.[†]

These registers do not need to be changed with every SD/MMC/CE-ATA command. Set them to a typical value according to the SD/MMC/CE-ATA specifications.

Related Information

- [Clock Setup](#) on page 14-46
Refer to this section for information on setting the clock source assignments.
- [Enumerated Card Stack](#) on page 14-43
Refer to this section for information on discovering the card stack according to the card type.

Enumerated Card Stack

The card stack performs the following tasks:

- Discovers the connected card[†]
- Sets the relative Card Address Register (RCA) in the connected card[†]
- Reads the card specific information[†]
- Stores the card specific information locally[†]

The card connected to the controller can be an MMC, CE-ATA, SD or SDIO (including IO ONLY, MEM ONLY and COMBO) card.

Identifying the Connected Card Type

To identify the connected card type, the following discovery sequence is needed:

1. Reset the card width 1 or 4 bit (`card_width2`) and card width 8 bit (`card_width1`) fields in the `ctype` register to 0.
2. Identify the card type as SD, MMC, SDIO or SDIO-COMBO:

- a. Send an SD/SDIO `IO_SEND_OP_COND` (CMD5) command with argument 0 to the card.
- b. Read `resp0` on the controller. The response to the `IO_SEND_OP_COND` command gives the voltage that the card supports.
- c. Send the `IO_SEND_OP_COND` command, with the desired voltage window in the arguments. This command sets the voltage window and makes the card exit the initialization state.
- d. Check bit 27 in `resp0`:
 - If bit 27 is 0, the SDIO card is IO ONLY. In this case, proceed to [step 5](#).
 - If bit 27 is 1, the card type is SDIO COMBO. Continue with the following steps.
3. Go to [Card Type is Either SDIO COMBO or Still in Initialization](#) on page 14-44.
4. Go to [Determine if Card is a CE-ATA 1.1, CE-ATA 1.0, or MMC Device](#) on page 14-45.
5. At this point, the software has determined the card type as SD/SDHC, SDIO or SDIO-COMBO. Now it must enumerate the card stack according to the type that has been discovered.
6. Set the card clock source frequency to the frequency of identification clock rate, 400 KHz. Use one of the following discovery command sequences:
 - For an SD card or an SDIO memory section, send the following SD/SDIO command sequence:
 - `GO_IDLE_STATE`
 - `SEND_IF_COND`
 - `SD_SEND_OP_COND` (ACMD41)
 - `ALL_SEND_CID` (CMD2)
 - `SEND_RELATIVE_ADDR` (CMD3)
 - For an SDIO card, send the following command sequence:
 - `IO_SEND_OP_COND`
 - If the function count is valid, send the `SEND_RELATIVE_ADDR` command.
 - For an MMC, send the following command sequence:
 - `GO_IDLE_STATE`
 - `SEND_OP_COND` (CMD1)
 - `ALL_SEND_CID`
 - `SEND_RELATIVE_ADDR`
7. You can change the card clock frequency after discovery by writing a value to the `clkdiv` register that divides down the `sdmmc_clk` clock.

The following list shows typical clock frequencies for various types of cards:

- SD memory card, 25 MHz[†]
- MMC card device, 12.5 MHz[†]
- Full speed SDIO, 25 MHz[†]
- Low speed SDIO, 400 kHz[†]

Related Information

[SD Association](#)

To learn more about how SD technology works, visit the SD Association website (www.sdcard.org).

Card Type is Either SDIO COMBO or Still in Initialization

Only continue with this step if the SDIO card type is COMBO or there is no response received from the previous `IO_SEND_OP_COND` command. Otherwise, skip to [step 5](#) of the *Identifying the Connected Card Type* section.

1. Send the SD/SDIO `SEND_IF_COND` (CMD8) command with the following arguments:

- Bit[31:12] = 0x0 (reserved bits)[†]
- Bit[11:8] = 0x1 (supply voltage value)[†]
- Bit[7:0] = 0xAA (preferred check pattern by SD memory cards compliant with SDIO Simplified Specification Version 2.00 and later.)[†]

Refer to *SDIO Simplified Specification Version 2.00* as described on the SD Association website.

- If a response is received to the previous SEND_IF_COND command, the card supports SD High-Capacity, compliant with *SD Specifications, Part 1, Physical Layer Simplified Specification Version 2.00*.
 - If no response is received, proceed to [the next decision statement](#).
2. Send the SD_SEND_OP_COND (ACMD41) command with the following arguments:
 - Bit[31] = 0x0 (reserved bits)[†]
 - Bit[30] = 0x1 (high capacity status)[†]
 - Bit[29:25] = 0x0 (reserved bits)[†]
 - Bit[24] = 0x1 (S18R --supports voltage switching for 1.8V)[†]
 - Bit[23:0] = supported voltage range[†]
 - If the previous SD_SEND_OP_COND command receives a response, then the card type is SDHC. Otherwise, the card is MMC or CE-ATA. In either case, skip the following steps and proceed to [step 5](#) of the "Identifying the Connected Card Type" section.
 - If the initial SEND_IF_COND command does not receive a response, then the card does not support High Capacity SD2.0. Now, proceed to [step step 3](#).
 3. Next, issue the GO_IDLE_STATE command followed by the SD_SEND_OP_COND command with the following arguments:
 - Bit[31] = 0x0 (reserved bits)[†]
 - Bit[30] = 0x0 (high capacity status)[†]
 - Bit[29:24] = 0x0 (reserved bits)[†]
 - Bit[23:0] = supported voltage range[†]

If a response is received to the previous SD_SEND_OP_COND command, the card is SD type. Otherwise, the card is MMC or CE-ATA.

Note: You must issue the SEND_IF_COND command prior to the first SD_SEND_OP_COND command, to initialize the High Capacity SD memory card. The card returns busy as a response to the SD_SEND_OP_COND command when any of the following conditions are true:

- The card executes its internal initialization process.
- A SEND_IF_COND command is not issued before the SD_SEND_OP_COND command.
- The ACMD41 command is issued. In the command argument, the Host Capacity Support (HCS) bit is set to 0, for a high capacity SD card.

Determine if Card is a CE-ATA 1.1, CE-ATA 1.0, or MMC Device

Use the following sequence to determine whether the card is a CE-ATA 1.1, CE-ATA 1.0, or MMC device:

Determine whether the card is a CE-ATA v1.1 card device by attempting to select ATA mode.

1. Send the SD/SDIO SEND_IF_COND command, querying byte 504 (S_CMD_SET) of the EXT_CSD register block in the external card.



If bit 4 is set to 1, the card device supports ATA mode.

2. Send the **SWITCH_FUNC** (CMD6) command, setting the ATA bit (bit 4) of the EXT_CSD register slice 191 (CMD_SET) to 1.

This command selects ATA mode and activates the ATA command set.

3. You can verify the currently selected mode by reading it back from byte 191 of the EXT_CSD register.
4. Skip to [step 5](#) of the *Identifying the Connected Card Type* section.

If the card device does not support ATA mode, it might be an MMC card or a CE-ATA v1.0 card.

Proceed to the [next section](#) to determine whether the card is a CE-ATA 1.0 card device or an MMC card device.

Determine whether the card is a CE-ATA 1.0 card device or an MMC card device by sending the RW_REG command.

If a response is received and the response data contains the CE-ATA signature, the card is a CE-ATA 1.0 card device. Otherwise, the card is an MMC card device.

Clock Setup

The following registers of the SD/MMC controller allow software to select the desired clock frequency for the card:

- `clksrc`
- `clkdiv`
- `clkena`

The controller loads these registers when it receives an update clocks command.

Changing the Card Clock Frequency

To change the card clock frequency, perform the following steps:

1. Before disabling the clocks, ensure that the card is not busy with any previous data command. To do so, verify that the `data_busy` bit of the status register (`status`) is 0.
2. Reset the `cclk_enable` bit of the `clkena` register to 0, to disable the card clock generation.
3. Reset the `clksrc` register to 0.
4. Set the following bits in the `cmd` register to 1:
 - `update_clk_regs_only`—Specifies the update clocks command[†]
 - `wait_prvdata_complete`—Ensures that clock parameters do not change until any ongoing data transfer is complete[†]
 - `start_cmd`—Initiates the command[†]
5. Wait until the `start_cmd` and `update_clk_regs_only` bits change to 0. There is no interrupt when the clock modification completes. The controller does not set the `command_done` bit in the `rintsts` register upon command completion. The controller might signal a hardware lock error if it already has another command in the queue. In this case, return to [Step 4](#).

For information about hardware lock errors, refer to the "Interrupt and Error Handling" chapter.

6. Reset the `sdmmc_clk_enable` bit to 0 in the `enable` register of the clock manager peripheral PLL group (`perpllgrp`).
7. In the control register (`ctrl`) of the SDMMC controller group (`sdmmcgrp`) in the system manager, set the drive clock phase shift select (`drvsel`) and sample clock phase shift select (`smp1sel`) bits to specify the required phase shift value.
8. Set the `sdmmc_clk_enable` bit in the `Enable` register of the clock manager `perpllgrp` group to 1.
9. Set the `clkdiv` register of the controller to the correct divider value for the required clock frequency.
10. Set the `cclk_enable` bit of the `clkena` register to 1, to enable the card clock generation.

You can also use the `clkena` register to enable low-power mode, which automatically stops the `sdmmc_clk_out` clock when the card is idle for more than eight clock cycles.

Related Information

[Interrupt and Error Handling](#) on page 14-73

Refer to this section for information about hardware lock errors.

Controller/DMA/FIFO Buffer Reset Usage

The following list shows the effect of reset on various parts in the SD/MMC controller:[†]

- Controller reset—resets the controller by setting the `controller_reset` bit in the `ctrl` register to 1. Controller reset resets the CIU and state machines, and also resets the BIU-to-CIU interface. Because this reset bit is self-clearing, after issuing the reset, wait until this bit changes to 0.[†]
- FIFO buffer reset—resets the FIFO buffer by setting the FIFO reset bit (`fifo_reset`) in the `ctrl` register to 1. FIFO buffer reset resets the FIFO buffer pointers and counters in the FIFO buffer. Because this reset bit is self-clearing, after issuing the reset, wait until this bit changes to 0.[†]
- DMA reset—resets the internal DMA controller logic by setting the DMA reset bit (`dma_reset`) in the `ctrl` register to 1, which immediately terminates any DMA transfer in progress. Because this reset bit is self-clearing, after issuing the reset, wait until this bit changes to 0.[†]

Note: Ensure that the DMA is idle before performing a DMA reset. Otherwise, the L3 interconnect might be left in an indeterminate state.[†]

Intel recommends setting the `controller_reset`, `fifo_reset`, and `dma_reset` bits in the `ctrl` register to 1 first, and then resetting the `rintsts` register to 0 using another write, to clear any resultant interrupt.

Enabling ECC

1. Turn on the ECC hardware, but disable the interrupts.
2. Initialize the SRAM in the peripheral.⁽⁴⁷⁾
3. Clear the ECC event bits, because these bits may have become asserted after Step 1.
4. Enable the ECC interrupts now that the ECC bits have been set.

Enabling FIFO Buffer ECC

To protect the FIFO buffer data with ECC, you must enable the ECC feature before performing any operations with the SD/MMC controller. Perform the following steps to enable the FIFO buffer ECC feature:

1. Verify there are no commands committed to the controller.
2. Ensure that the FIFO buffer is initialized. Initialize the FIFO buffer by writing 0 to all 1024 FIFO buffer locations. A FIFO buffer write to any address from 0x200 to the maximum FIFO buffer size is valid.
3. Set the SDMMC RAM ECC single and double, correctable error interrupt status bits (`serrporta`, `derrporta`, `serrportb`, and `derrportb`) to 1 in the `sdmmc` register in the `eccgrp` group of the system manager, to clear any previously-detected ECC errors.
4. Reset the FIFO buffer by setting the `fifo_reset` bit to 1 in the `ctrl` register. This action resets pointers and counters in the FIFO buffer. This reset bit is self-clearing, so after issuing the reset, wait until the bit changes to 0.
5. Set the `en` bit in `sdmmc` register in `eccgrp` group of the system manager to 1, to enable ECC for the FIFO buffer in SD/MMC controller.

⁽⁴⁷⁾ This is the case for the following peripherals: SD/MMC, NAND, On-Chip, DMA, USB, and EMAC.

Non-Data Transfer Commands

To send any non-data transfer command, the software needs to write the `cmd` register and the `cmdarg` register with appropriate parameters. Using these two registers, the controller forms the command and sends it to the `CMD` pin. The controller reports errors in the command response through the error bits of the `rintsts` register.[†]

When a response is received—either erroneous or valid—the controller sets the `command_done` bit in the `rintsts` register to 1. A short response is copied to `resp0`, while a long response is copied to all four response registers (`resp0`, `resp1`, `resp2`, and `resp3`).[†] For long responses, bit 31 of `resp3` represents the MSB and bit 0 of `resp0` represents the LSB.[†]

For basic and non-data transfer commands, perform the following steps:

1. Write the `cmdarg` register with the appropriate command argument parameter.[†]
2. Write the `cmd` register with the settings in *Register Settings for Non-Data Transfer Command*.[†]
3. Wait for the controller to accept the command. The `start_cmd` bit changes to 0 when the command is accepted.[†]

The following actions occur when the command is loaded into the controller:[†]

- If no previous command is being processed, the controller accepts the command for execution and resets the `start_cmd` bit in the `cmd` register to 0. If a previous command is being processed, the controller loads the new command in the command buffer.[†]
 - If the controller is unable to load the new command—that is, a command is already in progress, a second command is in the buffer, and a third command is attempted—the controller generates a hardware lock error.[†]
4. Check if there is a hardware lock error.[†]
 5. Wait for command execution to complete. After receiving either a response from a card or response timeout, the controller sets the `command_done` bit in the `rintsts` register to 1. Software can either poll for this bit or respond to a generated interrupt (if enabled).[†]
 6. Check if the response timeout boot acknowledge received (`bar`), `rcrc`, or `re` bit is set to 1. Software can either respond to an interrupt raised by these errors or poll the `re`, `rcrc`, and `bar` bits of the `rintsts` register. If no response error is received, the response is valid. If required, software can copy the response from the response registers.[†]

Note: Software cannot modify clock parameters while a command is being executed.[†]

Related Information

[cmd Register Settings for Non-Data Transfer Command[†]](#) on page 14-48

Refer to this table for information about Non-Data Transfer commands.

cmd Register Settings for Non-Data Transfer Command[†]

Table 14-20: Default

Parameter	Value	Comment
<code>start_cmd</code>	1	This bit resets itself to 0 after the command is committed.
<code>use_hold_reg</code>	1 or 0	Choose the value based on the speed mode used.

Parameter	Value	Comment
update_clk_regs_only	0	Indicates that the command is not a clock update command
data_expected	0	Indicates that the command is not a data command
card_number	1	For one card
cmd_index	Command Index	Set this parameter to the command number. For example, set to 8 for the SD/SDIO SEND_IF_COND (CMD8) command.
send_initialization	0 or 1	1 for card reset commands such as the SD/SDIO GO_IDLE_STATE command 0 otherwise
stop_abort_cmd	0 or 1	1 for a command to stop data transfer, such as the SD/SDIO STOP_TRANSMISSION command 0 otherwise
response_length	0 or 1	1 for R2 (long) response 0 for short response
response_expect	0 or 1	0 for commands with no response, such as SD/SDIO GO_IDLE_STATE, SET_DSR (CMD4), or GO_INACTIVE_STATE (CMD15). 1 otherwise

Table 14-21: User Selectable

Parameter	Value	Comment
wait_prvdata_complete	1	Before sending a command on the command line, the host must wait for completion of any data command already in process. Intel recommends that you set this bit to 1, unless the current command is to query status or stop data transfer when transfer is in progress.
check_response_crc	1 or 0	1 if the response includes a valid CRC, and the software is required to crosscheck the response CRC bits. 0 otherwise

Data Transfer Commands

Data transfer commands transfer data between the memory card and the controller. To issue a data command, the controller requires a command argument, total data size, and block size. Data transferred to or from the memory card is buffered by the controller FIFO buffer.[†]

Confirming Transfer State

Before issuing a data transfer command, software must confirm that the card is not busy and is in a transfer state, by performing the following steps:[†]

1. Issue an SD/SDIO SEND_STATUS (CMD13) command. The controller sends the status of the card as the response to the command.[†]
2. Check the card's busy status.[†]
3. Wait until the card is not busy.[†]
4. Check the card's transfer status. If the card is in the stand-by state, issue an SD/SDIO SELECT/DESELECT_CARD (CMD7) command to place it in the transfer state.[†]

Busy Signal After CE-ATA RW_BLK Write Transfer

During CE-ATA RW_BLK write transfers, the MMC busy signal might be asserted after the last block. If the CE-ATA card device interrupt is disabled (the nIEN bit in the card device's ATA control register is set to 1), the dto bit in the rintsts register is set to 1 even though the card sends MMC BUSY. The host cannot issue the CMD60 command to check the ATA busy status after a CMD61 command. Instead, the host must perform one of the following actions:[†]

- Issue the SEND_STATUS command and check the MMC busy status before issuing a new CMD60 command.[†]
- Issue the CMD39 command and check the ATA busy status before issuing a new CMD60 command.[†]

For the data transfer commands, software must set the ctype register to the bus width that is programmed in the card.[†]

Data Transfer Interrupts

The controller generates an interrupt for different conditions during data transfer, which are reflected in the following rintsts register bits:[†]

1. dto—Data transfer is over or terminated. If there is a response timeout error, the controller does not attempt any data transfer and the Data Transfer Over bit is never set.[†]
2. Transmit FIFO data request bit (txdr)—The FIFO buffer threshold for transmitting data is reached; software is expected to write data, if available, into the FIFO buffer.[†]
3. Receive FIFO data request bit (rxdr)—The FIFO buffer threshold for receiving data is reached; software is expected to read data from the FIFO buffer.[†]
4. hto—The FIFO buffer is empty during transmission or is full during reception. Unless software corrects this condition by writing data for empty condition, or reading data for full condition, the controller cannot continue with data transfer. The clock to the card is stopped.[†]
5. bds—The card has not sent data within the timeout period.[†]
6. dcrc—A CRC error occurred during data reception.[†]
7. sbe—The start bit is not received during data reception.[†]
8. ebe—The end bit is not received during data reception, or for a write operation. A CRC error is indicated by the card.[†]

dcrc, sbe, and ebe indicate that the received data might have errors. If there is a response timeout, no data transfer occurs.[†]

Single-Block or Multiple-Block Read

To implement a single-block or multiple-block read, the software performs the following steps:[†]

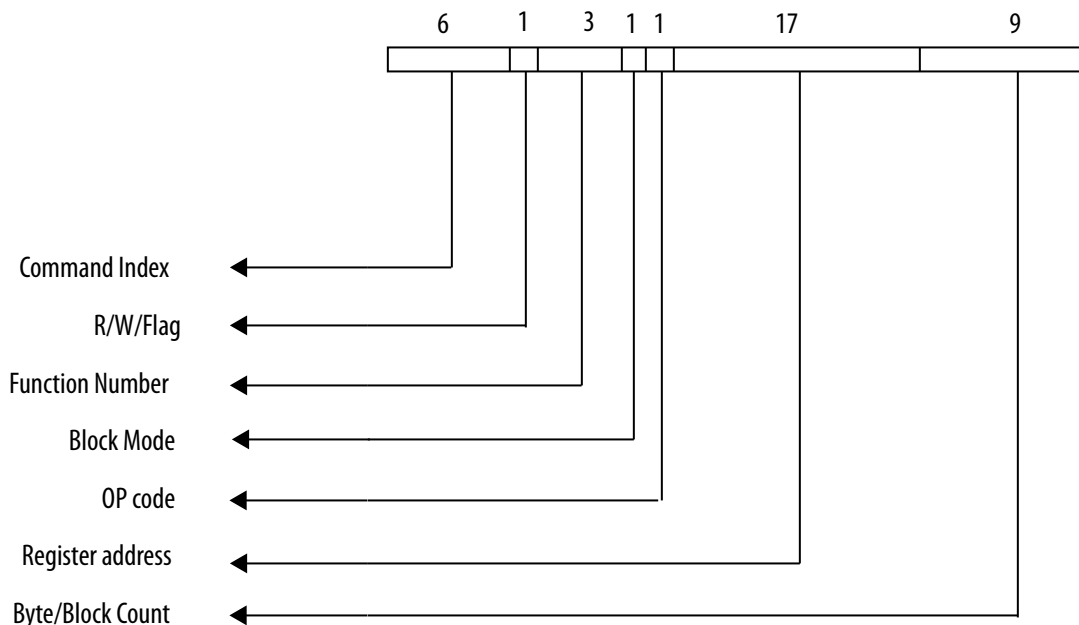
1. Write the data size in bytes to the `bytcnt` register. For a multi-block read, `bytcnt` must be a multiple of the block size.[†]
2. Write the block size in bytes to the `blksiz` register. The controller expects data to return from the card in blocks of size `blksiz`.[†]
3. If the read round trip delay, including the card delay, is greater than half of `sdmmc_clk_divided`, write to the card threshold control register (`cardthrctl`) to ensure that the card clock does not stop in the middle of a block of data being transferred from the card to the host. For more information, refer to *Card Read Threshold*.[†]

Note: If the card read threshold enable bit (`cardrdthren`) is 0, the host system must ensure that the RX FIFO buffer does not become full during a read data transfer by ensuring that the RX FIFO buffer is read at a rate faster than that at which data is written into the FIFO buffer. Otherwise, an overflow might occur.[†]

4. Write the `cmdarg` register with the beginning data address for the data read.[†]
5. Write the `cmd` register with the parameters listed in *cmd Register Settings for Single-Block and Multiple-Block Reads*. For SD and MMC cards, use the SD/SDIO `READ_SINGLE_BLOCK` (CMD17) command for a single-block read and the `READ_MULTIPLE_BLOCK` (CMD18) command for a multiple-block read. For SDIO cards, use the `IO_RW_EXTENDED` (CMD53) command for both single-block and multiple-block transfers. The command argument for (CMD53) is shown in the figure, below. After writing to the `cmd` register, the controller starts executing the command. When the command is sent to the bus, the Command Done interrupt is generated.[†]
6. Software must check for data error interrupts, reported in the `dcrc`, `bds`, `sbe`, and `ebe` bits of the `rintsts` register. If required, software can terminate the data transfer by sending an SD/SDIO STOP command.[†]
7. Software must check for host timeout conditions in the `rintsts` register:[†]
 - Receive FIFO buffer data request[†]
 - Data starvation from host—the host is not reading from the FIFO buffer fast enough to keep up with data from the card. To correct this condition, software must perform the following steps:[†]
 - Read the `fifo_count` field of the status register[†]
 - Read the corresponding amount of data out of the FIFO buffer[†]

In both cases, the software must read data from the FIFO buffer and make space in the FIFO buffer for receiving more data.[†]

8. When a DTO interrupt is received, the software must read the remaining data from the FIFO buffer.[†]

Figure 14-15: Command Argument for IO_RW_EXTENDED (CMD53)[†]**Related Information**

- [Card Read Threshold](#) on page 14-70
Refer to this section for information about the thresholds for a card read.
- [cmd Register Settings for Single-Block and Multiple-Block Reads[†]](#) on page 14-52
Refer to this table for information about the settings for Single-Block and Multiple-Block Reads.

cmd Register Settings for Single-Block and Multiple-Block Reads[†]

Table 14-22: cmd Register Settings for Single-Block and Multiple-Block Reads (Default)

Parameter	Value	Comment
start_cmd	1	This bit resets itself to 0 after the command is committed.
use_hold_reg	1 or 0	Choose the value based on speed mode used.
update_clk_regs_only	0	Does not need to update clock parameters
data_expected	1	Data command
card_number	1	For one card
transfer_mode	0	Block transfer

Parameter	Value	Comment
send_initialization	0	1 for a card reset command such as the SD/SDIO GO_IDLE_STATE command 0 otherwise
stop_abort_cmd	0	1 for a command to stop data transfer such as the SD/SDIO STOP_TRANSMISSION command 0 otherwise
send_auto_stop	0 or 1	Refer to <i>Auto Stop</i> for information about how to set this parameter.
read_write	0	Read from card
response_length	0	1 for R2 (long) response 0 for short response
response_expect	1 or 0	0 for commands with no response, such as SD/SDIO GO_IDLE_STATE, SET_DSR, and GO_INACTIVE_STATE. 1 otherwise

Table 14-23: cmd Register Settings for Single-Block and Multiple-Block Reads (User Selectable)

Parameter	Value	Comment
wait_prvdata_complete	1 or 0	0 - sends command to CIU immediately 1 - sends command after previous data transfer ends
check_response_crc	1 or 0	0 - Controller must not check response CRC 1 - Controller must check response CRC
cmd_index	Command Index	Set this parameter to the command number. For example, set to 17 or 18 for SD/SDIO READ_SINGLE_BLOCK (CMD17) or READ_MULTIPLE_BLOCK (CMD18)

Related Information[Auto-Stop](#) on page 14-29

Refer to this table for information about setting the send_auto_stop parameter.

Single-Block or Multiple-Block Write

The following steps comprise a single-block or multiple-block write:



1. Write the data size in bytes to the `bytcnt` register. For a multi-block write, `bytcnt` must be a multiple of the block size.[†]
2. Write the block size in bytes to the `blksiz` register. The controller sends data in blocks of size `blksiz` each.[†]
3. Write the `cmdarg` register with the data address to which data must be written.[†]
4. Write data into the FIFO buffer. For best performance, the host software should write data continuously until the FIFO buffer is full.[†]
5. Write the `cmd` register with the parameters listed in *cmd Register Settings for Single-Block and Multiple-Block Write*. For SD and MMC cards, use the SD/SDIO `WRITE_BLOCK` (CMD24) command for a single-block write and the `WRITE_MULTIPLE_BLOCK` (CMD25) command for a multiple-block writes. For SDIO cards, use the `IO_RW_EXTENDED` command for both single-block and multiple-block transfers.[†]

After writing to the `cmd` register, the controller starts executing a command if there is no other command already being processed. When the command is sent to the bus, a Command Done interrupt is generated.[†]

6. Software must check for data error interrupts; that is, for `dcrc`, `bds`, and `ebe` bits of the `rintsts` register. If required, software can terminate the data transfer early by sending the SD/SDIO `STOP` command.[†]
7. Software must check for host timeout conditions in the `rintsts` register:[†]
 - Transmit FIFO buffer data request.[†]
 - Data starvation by the host—the controller wrote data to the card faster than the host could supply the data.[†]

In both cases, the software must write data into the FIFO buffer.[†]

There are two types of transfers: open-ended and fixed length.[†]

- Open-ended transfers—For an open-ended block transfer, the byte count is 0. At the end of the data transfer, software must send the `STOP_TRANSMISSION` command (CMD12).[†]
- Fixed-length transfers—The byte count is nonzero. You must already have written the number of bytes to the `bytcnt` register. The controller issues the `STOP` command for you if you set the `send_auto_stop` bit of the `cmd` register to 1. After completion of a transfer of a given number of bytes, the controller sends the `STOP` command. Completion of the `AUTO_STOP` command is reflected by the Auto Command Done interrupt. A response to the `AUTO_STOP` command is written to the `resp1` register. If software does not set the `send_auto_stop` bit in the `cmd` register to 1, software must issue the `STOP` command just like in the open-ended case.[†]

When the `dto` bit of the `rintsts` register is set, the data command is complete.[†]

cmd Register Settings for Single-Block and Multiple-Block Write

Table 14-24: cmd Register Settings for Single-Block and Multiple-Block Write (Default)[†]

Parameter	Value	Comment
<code>start_cmd</code>	1	This bit resets itself to 0 after the command is committed (accepted by the BIU).
<code>use_hold_reg</code>	1 or 0	Choose the value based on speed mode used.
<code>update_clk_regs_only</code>	0	Does not need to update clock parameters

Parameter	Value	Comment
data_expected	1	Data command
card_number	1	For one card
transfer_mode	0	Block transfer
send_initialization	0	Can be 1, but only for card reset commands such as SD/SDIO GO_IDLE_STATE
stop_abort_cmd	0	Can be 1 for commands to stop data transfer such as SD/SDIO STOP_TRANSMISSION
send_auto_stop	0 or 1	Refer to <i>Auto Stop</i> for information about how to set this parameter.
read_write	1	Write to card
response_length	0	Can be 1 for R2 (long) responses
response_expect	1	Can be 0 for commands with no response. For example, SD/SDIO GO_IDLE_STATE, SET_DSR, GO_INACTIVE_STATE etc.

Table 14-25: cmd Register Settings for Single-Block and Multiple-Block Write (User Selectable)[†]

Parameter	Value	Comment
wait_prvdata_complete	1	0—Sends command to the CIU immediately 1—Sends command after previous data transfer ends
check_response_crc	1	0—Controller must not check response CRC 1—Controller must check response CRC
cmd_index	Command Index	Set this parameter to the command number. For example, set to 24 for SD/SDIO WRITE_BLOCK (CMD24) or 25 for WRITE_MULTIPLE_BLOCK (CMD25).

Related Information[Auto-Stop](#) on page 14-29

Refer to this table for information about setting the send_auto_stop parameter.

Stream Read and Write

In a stream transfer, if the byte count is equal to 0, the software must also send the SD/SDIO STOP command. If the byte count is not 0, when a given number of bytes completes a transfer, the controller sends the STOP command automatically. Completion of this AUTO_STOP command is reflected by the

Auto_command_done interrupt. A response to an AUTO_STOP command is written to the resp1 register. A stream transfer is allowed only for card interfaces with a 1-bit data bus.[†]

A stream read requires the same steps as the block read described in *Single-Block or Multiple-Block Read*, except for the following bits in the cmd register:[†]

- transfer_mode = 0x1 (for stream transfer)[†]
- cmd_index = 20 (SD/SDIO CMD20)[†]

A stream write requires the same steps as the block write mentioned in *Single-Block or Multiple-Block Write*, except for the following bits in the cmd register:[†]

- transfer_mode = 0x1 (for stream transfer)[†]
- cmd_index = 11 (SD/SDIO CMD11)[†]

Related Information

- [Single-Block or Multiple-Block Read](#) on page 14-51
Refer to this section for more information about a stream read.
- [Single-Block or Multiple-Block Write](#) on page 14-53
Refer to this section for more information about a stream write.

Packed Commands

To reduce overhead, read and write commands can be packed in groups of commands—either all read or all write—that transfer the data for all commands in the group in one transfer on the bus. Use the SD/SDIO SET_BLOCK_COUNT (CMD23) command to state ahead of time how many blocks are transferred. Then issue a single READ_MULTIPLE_BLOCK or WRITE_MULTIPLE_BLOCK command to read or write multiple blocks.[†]

- SET_BLOCK_COUNT—set block count (number of blocks transferred using the READ_MULTIPLE_BLOCK or WRITE_MULTIPLE_BLOCK command) [†]
- READ_MULTIPLE_BLOCK—multiple-block read command [†]
- WRITE_MULTIPLE_BLOCK—multiple-block write command[†]

Packed commands are organized in packets by the application software and are transparent to the controller.[†]

Related Information

[JEDEC Global Standards of the Microelectronics Industry](#)

For more information about packed commands, refer to JEDEC Standard No. 84-A441, available on the JEDEC website.

Transfer Stop and Abort Commands

This section describes stop and abort commands. The SD/SDIO STOP_TRANSMISSION command can terminate a data transfer between a memory card and the controller. The ABORT command can terminate an I/O data transfer for only an SDIO card.[†]

STOP_TRANSMISSION (CMD12)

The host can send the STOP_TRANSMISSION (CMD12) command on the CMD pin at any time while a data transfer is in progress. Perform the following steps to send the STOP_TRANSMISSION command to the SD/SDIO card device:[†]

1. Set the `wait_prvdata_complete` bit of the `cmd` register to 0.[†]
2. Set the `stop_abort_cmd` in the `cmd` register to 1, which ensures that the CIU stops.[†]

The `STOP_TRANSMISSION` command is a non-data transfer command.[†]

Related Information

[Non-Data Transfer Commands](#) on page 14-48

Refer to this section for information on the `STOP_TRANSMISSION` command.

ABORT

The `ABORT` command can only be used with SDIO cards. To abort the function that is transferring data, program the `ABORT` function number in the `ASx[2:0]` bits at address 0x06 of the card common control register (CCCR) in the card device, using the `IO_RW_DIRECT` (CMD52) command. The CCCR is located at the base of the card space 0x00 – 0xFF.[†]

Note: The `ABORT` command is a non-data transfer command.[†]

Related Information

[Non-Data Transfer Commands](#) on page 14-48

Refer to this section for information on the `ABORT` command.

Sending the ABORT Command

Perform the following steps to send the `ABORT` command to the SDIO card device:[†]

1. Set the `cmdarg` register to include the appropriate command argument parameters listed in *cmdarg Register Settings for SD/SDIO ABORT Command*.[†]
2. Send the `IO_RW_DIRECT` command by setting the following fields of the `cmd` register:[†]
 - Set the command index to 0x52 (`IO_RW_DIRECT`).[†]
 - Set the `stop_abort_cmd` bit of the `cmd` register to 1 to inform the controller that the host aborted the data transfer.[†]
 - Set the `wait_prvdata_complete` bit of the `cmd` register to 0.[†]
3. Wait for the `cmd` bit in the `rintsts` register to change to 1.[†]
4. Read the response to the `IO_RW_DIRECT` command (R5) in the response registers for any errors.[†]

For more information about response values, refer to the *Physical Layer Simplified Specification*, Version 3.01, available on the SD Association website.

Related Information

[SD Association](#)

To learn more about how SD technology works, visit the SD Association website (www.sdcard.org).

cmdarg Register Settings for SD/SDIO ABORT Command[†]

Table 14-26: `cmdarg` Register Settings for SD/SDIO ABORT Command

Bits	Contents	Value
31	R/W flag	1
30:28	Function number	0, for access to the CCCR in the card device

Bits	Contents	Value
27	RAW flag	1, if needed to read after write
26	Don't care	-
25:9	Register address	0x06
8	Don't care	-
7:0	Write data	Function number to abort

Internal DMA Controller Operations

For better performance, you can use the internal DMA controller to transfer data between the host and the controller. This section describes the internal DMA controller's initialization process, and transmission sequence, and reception sequence.

Internal DMA Controller Initialization

To initialize the internal DMA controller, perform the following steps:[†]

- Set the required bmod register bits:[†]
 - If the internal DMA controller enable bit (de) of the bmod register is set to 0 during the middle of a DMA transfer, the change has no effect. Disabling only takes effect for a new data transfer command.[†]
 - Issuing a software reset immediately terminates the transfer. Prior to issuing a software reset, Intel recommends the host reset the DMA interface by setting the dma_reset bit of the ctrl register to 1.[†]
 - The pbl field of the bmod register is read-only and a direct reflection of the contents of the DMA multiple transaction size field (dw_dma_multiple_transaction_size) in the fifoth register.[†]
 - The fb bit of the bmod register has to be set appropriately for system performance.[†]
- Write to the idinten register to mask unnecessary interrupt causes according to the following guidelines:[†]
 - When a Descriptor Unavailable interrupt is asserted, the software needs to form the descriptor, appropriately set its own bit, and then write to the poll demand register (pldmnd) for the internal DMA controller to re-fetch the descriptor.[†]
 - It is always appropriate for the software to enable abnormal interrupts because any errors related to the transfer are reported to the software.[†]
- Populate either a transmit or receive descriptor list in memory. Then write the base address of the first descriptor in the list to the internal DMA controller's descriptor list base address register (dbaddr). The DMA controller then proceeds to load the descriptor list from memory. *Internal DMA Controller Transmission Sequences* and *Internal DMA Controller Reception Sequences* describe this step in detail.[†]

Related Information

- [Internal DMA Controller Transmission Sequences](#) on page 14-59
Refer to this section for information about the Internal DMA Controller Transmission Sequences.
- [Internal DMA Controller Reception Sequences](#) on page 14-59
Refer to this section for information about the Internal DMA Controller Reception Sequences.

Internal DMA Controller Transmission Sequences

To use the internal DMA controller to transmit data, perform the following steps:

1. The host sets up the Descriptor fields (DES0—DES3) for transmission and sets the OWN bit (DES0[31]) to 1. The host also loads the data buffer in system memory with the data to be written to the SD card.[†]
2. The host writes the appropriate write data command (SD/SDIO WRITE_BLOCK or WRITE_MULTIPLE_BLOCK) to the cmd register. The internal DMA controller determines that a write data transfer needs to be performed.[†]
3. The host sets the required transmit threshold level in the tx_wmark field in the fifoth register.[†]
4. The internal DMA controller engine fetches the descriptor and checks the OWN bit. If the OWN bit is set to 0, the host owns the descriptor. In this case, the internal DMA controller enters the suspend state and asserts the Descriptor Unable interrupt. The host then needs to set the descriptor OWN bit to 1 and release the DMA controller by writing any value to the pldmnd register.[†]
5. The host must write the descriptor base address to the dbaddr register.[†]
6. The internal DMA controller waits for the Command Done (CD) bit in the rintsts register to be set to 1, with no errors from the BIU. This condition indicates that a transfer can be done.[†]
7. The internal DMA controller engine waits for a DMA interface request from BIU. The BIU divides each transfer into smaller chunks. Each chunk is an internal request to the DMA. This request is generated based on the transmit threshold value.[†]
8. The internal DMA controller fetches the transmit data from the data buffer in the system memory and transfers the data to the FIFO buffer in preparation for transmission to the card.[†]
9. When data spans across multiple descriptors, the internal DMA controller fetches the next descriptor and continues with its operation with the next descriptor. The Last Descriptor bit in the descriptor DES0 field indicates whether the data spans multiple descriptors or not.[†]
10. When data transmission is complete, status information is updated in the idsts register by setting the ti bit to 1, if enabled. Also, the OWN bit is set to 0 by the DMA controller by updating the DES0 field of the descriptor.[†]

Internal DMA Controller Reception Sequences

To use the internal DMA controller to receive data, perform the following steps:

1. The host sets up the descriptor fields (DES0—DES3) for reception and sets the OWN (DES0 [31]) to 1.[†]
2. The host writes the read data command to the cmd register in BIU. The internal DMA controller determines that a read data transfer needs to be performed.[†]
3. The host sets the required receive threshold level in the rx_wmark field in the fifoth register.[†]
4. The internal DMA controller engine fetches the descriptor and checks the OWN bit. If the OWN bit is set to 0, the host owns the descriptor. In this case, the internal DMA controller enters suspend state and asserts the Descriptor Unable interrupt. The host then must set the descriptor OWN bit to 1 and release the DMA controller by writing any value to the pldmnd register.[†]
5. The host must write the descriptor base address to the dbaddr register.[†]
6. The internal DMA controller waits for the CD bit in the rintsts register to be set to 1, with no errors from the BIU. This condition indicates that a transfer can be done.[†]
7. The internal DMA controller engine waits for a DMA interface request from the BIU. The BIU divides each transfer into smaller chunks. Each chunk is an internal request to the DMA. This request is generated based on the receive threshold value.[†]

8. The internal DMA controller fetches the data from the FIFO buffer and transfers the data to system memory.[†]
9. When data spans across multiple descriptors, the internal DMA controller fetches the next descriptor and continues with its operation with the next descriptor. The Last Descriptor bit in the descriptor indicates whether the data spans multiple descriptors or not.[†]
10. When data reception is complete, status information is updated in the `idsts` register by setting the `ri` bit to 1, if enabled. Also, the `OWN` bit is set to 0 by the DMA controller by updating the `DES0` field of the descriptor.[†]

Commands for SDIO Card Devices

This section describes the commands to temporarily halt the transfers between the controller and SDIO card device.

Suspend and Resume Sequence

For SDIO cards, a data transfer between an I/O function and the controller can be temporarily halted using the **SUSPEND** command. This capability might be required to perform a high-priority data transfer with another function. When desired, the suspended data transfer can be resumed using the **RESUME** command.[†]

The **SUSPEND** and **RESUME** operations are implemented by writing to the appropriate bits in the **CCCR** (Function 0) of the SDIO card. To read from or write to the **CCCR**, use the controller's **IO_RW_DIRECT** command.[†]

Suspend

To suspend data transfer, perform the following steps:[†]

1. Check if the SDIO card supports the **SUSPEND/RESUME** protocol by reading the **SBS** bit in the **CCCR** at offset 0x08 of the card.[†]
2. Check if the data transfer for the required function number is in process. The function number that is currently active is reflected in the function select bits (**FSx**) of the **CCCR**, bits 3:0 at offset 0x0D of the card.[†]

Note: If the bus status bit (**BS**), bit 0 at address 0xC, is 1, only the function number given by the **FSx** bits is valid.[†]

3. To suspend the transfer, set the bus release bit (**BR**), bit 2 at address 0xC, to 1.[†]
4. Poll the **BR** and **BS** bits of the **CCCR** at offset 0x0C of the card until they are set to 0. The **BS** bit is 1 when the currently-selected function is using the data bus. The **BR** bit remains 1 until the bus release is complete. When the **BR** and **BS** bits are 0, the data transfer from the selected function is suspended.[†]
5. During a read-data transfer, the controller can be waiting for the data from the card. If the data transfer is a read from a card, the controller must be informed after the successful completion of the **SUSPEND** command. The controller then resets the data state machine and comes out of the wait state. To accomplish this, set the abort read data bit (`abort_read_data`) in the `ctrl` register to 1.[†]
6. Wait for data completion, by polling until the `dto` bit is set to 1 in the `rintsts` register. To determine the number of pending bytes to transfer, read the transferred CIU card byte count (`tcbcnt`) register of the controller. Subtract this value from the total transfer size. You use this number to resume the transfer properly.[†]

Resume

To resume the data transfer, perform the following steps:[†]

1. Check that the card is not in a transfer state, which confirms that the bus is free for data transfer.[†]
2. If the card is in a disconnect state, select it using the SD/SDIO SELECT/DESELECT_CARD command. The card status can be retrieved in response to an IO_RW_DIRECT or IO_RW_EXTENDED command.[†]
3. Check that a function to be resumed is ready for data transfer. Determine this state by reading the corresponding RF<n> flag in CCCR at offset 0x0F of the card. If RF<n> = 1, the function is ready for data transfer.[†]

Note: For detailed information about the RF<n> flags, refer to SDIO Simplified Specification Version 2.00, available on the SD Association website.[†]

4. To resume transfer, use the IO_RW_DIRECT command to write the function number at the FSx bits in the CCCR, bits 3:0 at offset 0x0D of the card. Form the command argument for the IO_RW_DIRECT command and write it to the cmdarg register. Bit values are listed in the following table.[†]

Table 14-27: cmdarg Bit Values for RESUME Command[†]

Bits	Content	Value
31	R/W flag	1
30:28	Function number	0, for CCCR access
27	RAW flag	1, read after write
26	Don't care	-
25:9	Register address	0x0D
8	Don't care	-
7:0	Write data	Function number that is to be resumed

5. Write the block size value to the blksize register. Data is transferred in units of this block size.[†]
6. Write the byte count value to the bytcnt register. Specify the total size of the data that is the remaining bytes to be transferred. It is the responsibility of the software to handle the data.[†]

To determine the number of pending bytes to transfer, read the transferred CIU card byte count register (tcbcnt). Subtract this value from the total transfer size to calculate the number of remaining bytes to transfer.[†]

7. Write to the cmd register similar to a block transfer operation. When the cmd register is written, the command is sent and the function resumes data transfer. For more information, refer to *Single-Block or Multiple-Block Read* and *Single-Block or Multiple-Block Write*.[†]
8. Read the resume data flag (DF) of the SDIO card device. Interpret the DF flag as follows:[†]
 - DF=1—The function has data for the transfer and begins a data transfer as soon as the function or memory is resumed.[†]
 - DF=0—The function has no data for the transfer. If the data transfer is a read, the controller waits for data. After the data timeout period, it issues a data timeout error.[†]

Related Information

- **SD Association**

To learn more about how SD technology works, visit the SD Association website (www.sdcard.org).

- [Single-Block or Multiple-Block Read](#) on page 14-51
Refer to this section for more information about writing to the cmd register.
- [Single-Block or Multiple-Block Write](#) on page 14-53
Refer to this section for more information about writing to the cmd register.

Read-Wait Sequence

Read_wait is used with SDIO cards only. It temporarily stalls the data transfer, either from functions or memory, and allows the host to send commands to any function within the SDIO card device. The host can stall this transfer for as long as required. The controller provides the facility to signal this stall transfer to the card.[†]

Signaling a Stall

To signal the stall, perform the following steps:[†]

1. Check if the card supports the read_wait facility by reading the SDIO card's SRW bit, bit 2 at offset 0x8 in the CCCR.[†]
2. If this bit is 1, all functions in the card support the read_wait facility. Use the SD/SDIO IO_RW_DIRECT command to read this bit.[†]
3. If the card supports the read_wait signal, assert it by setting the read wait bit (read_wait) in the ctrl register to 1.[†]
4. Reset the read_wait bit to 0 in the ctrl register.[†]

CE-ATA Data Transfer Commands

This section describes CE-ATA data transfer commands.

Related Information

[Data Transfer Commands](#) on page 14-50

Refer to this section for information about the basic settings and interrupts generated for different conditions.

ATA Task File Transfer Overview

ATA task file registers are mapped to addresses 0x00h through 0x10h in the MMC register space. The RW_REG command is used to issue the ATA command, and the ATA task file is transmitted in a single RW_REG MMC command sequence.[†]

The host software stack must write the task file image to the FIFO buffer before setting the cmdarg and cmd registers in the controller. The host processor then writes the address and byte count to the cmdarg register before setting the cmd register bits.[†]

For the RW_REG command, there is no CCS from the CE-ATA card device.[†]

ATA Task File Transfer Using the RW_MULTIPLE_REGISTER (RW_REG) Command

This command involves data transfer between the CE-ATA card device and the controller. To send a data command, the controller needs a command argument, total data size, and block size. Software receives or sends data through the FIFO buffer.[†]

Implementing ATA Task File Transfer

To implement an ATA task file transfer (read or write), perform the following steps:[†]

1. Write the data size in bytes to the `bytcnt` register. `bytcnt` must equal the block size, because the controller expects a single block transfer.[†]
2. Write the block size in bytes to the `blksiz` register.[†]
3. Write the `cmdarg` register with the beginning register address.[†]

You must set the `cmdarg`, `cmd`, `blksiz`, and `bytcnt` registers according to the tables in *Register Settings for ATA Task File Transfer*.[†]

Related Information

[Register Settings for ATA Task File Transfer](#) on page 14-63

Refer to this table for information on how to set these registers.

Register Settings for ATA Task File Transfer

Table 14-28: `cmdarg` Register Settings for ATA Task File Transfer[†]

Bit	Value	Comment
31	1 or 0	Set to 0 for read operation or set to 1 for write operation
30:24	0	Reserved (bits set to 0 by host processor)
23:18	0	Starting register address for read or write (DWORD aligned)
17:16	0	Register address (DWORD aligned)
15:8	0	Reserved (bits set to 0 by host processor)
7:2	16	Number of bytes to read or write (integral number of DWORD)
1:0	0	Byte count in integral number of DWORD

Table 14-29: `cmd` Register Settings for ATA Task File Transfer[†]

Bit	Value	Comment
<code>start_cmd</code>	1	
<code>ccs_expected</code>	0	CCS is not expected
<code>read_ceata_device</code>	0 or 1	Set to 1 if <code>RW_BLK</code> or <code>RW_REG</code> read
<code>update_clk_regs_only</code>	0	No clock parameters update command
<code>card_num</code>	0	
<code>send_initialization</code>	0	No initialization sequence
<code>stop_abort_cmd</code>	0	

Bit	Value	Comment
send_auto_stop	0	
transfer_mode	0	Block transfer mode. Block size and byte count must match number of bytes to read or write
read_write	1 or 0	1 for write and 0 for read
data_expected	1	Data is expected
response_length	0	
response_expect	1	
cmd_index	Command index	Set this parameter to the command number. For example, set to 24 for SD/SDIO WRITE_BLOCK (CMD24) or 25 for WRITE_MULTIPLE_BLOCK (CMD25).
wait_prvdata_complete	1	0 for send command immediately 1 for send command after previous DTO interrupt
check_response_crc	1	0 for not checking response CRC 1 for checking response CRC

Table 14-30: blksiz Register Settings for ATA Task File Transfer[†]

Bit	Value	Comment
31:16	0	Reserved bits set to 0
15:0 (block_size)	16	For accessing entire task file (16, 8-bit registers). Block size of 16 bytes

Table 14-31: bytcnt Register Settings for ATA Task File Transfer

Bit	Value	Comment
31:0	16	For accessing entire task file (16, 8-bit registers). Byte count value of 16 is used with the block size set to 16.

Reset and Card Device Discovery Overview

Before starting any CE-ATA operations, the host must perform a MMC reset and initialization procedure. The host and card device must negotiate the MMC transfer (MMC TRAN) state before the card enters the MMC TRAN state.[†]

The host must follow the existing MMC discovery procedure to negotiate the MMC TRAN state. After completing normal MMC reset and initialization procedures, the host must query the initial ATA task file values using the RW_REG or CMD39 command.[†]

By default, the MMC block size is 512 bytes—indicated by bits 1:0 of the srcControl register inside the CE-ATA card device. The host can negotiate the use of a 1 KB or 4 KB MMC block sizes. The card indicates MMC block sizes that it can support through the srcCapabilities register in the MMC; the host reads this register to negotiate the MMC block size. Negotiation is complete when the host controller writes the MMC block size into the srcControl register bits 1:0 of the card.[†]

Related Information

[JEDEC Global Standards of the Microelectronics Industry](#)

For information about the (MMC TRAN) state, MMC reset and initialization, refer to JEDEC Standard No. 84-A441, available on the JEDEC website.

ATA Payload Transfer Using the RW_MULTIPLE_BLOCK (RW_BLK) Command

This command involves data transfer between the CE-ATA card device and the controller. To send a data command, the controller needs a command argument, total data size, and block size. Software receives or sends data through the FIFO buffer.[†]

Implementing ATA Payload Transfer

To implement an ATA payload transfer (read or write), perform the following steps:[†]

1. Write the data size in bytes to the bytcnt register.[†]
2. Write the block size in bytes to the blksiz register. The controller expects a single/multiple block transfer.[†]
3. Write to the cmdarg register to indicate the data unit count.[†]

Register Settings for ATA Payload Transfer

You must set the cmdarg, cmd, blksiz, and bytcnt registers according to the following tables.[†]

Table 14-32: cmdarg Register Settings for ATA Payload Transfer[†]

Bits	Value	Comment
31	1 or 0	Set to 0 for read operation or set to 1 for write operation
30:24	0	Reserved (bits set to 0 by host processor)
23:16	0	Reserved (bits set to 0 by host processor)
15:8	Data count	Data Count Unit [15:8]
7:0	Data count	Data Count Unit [7:0]

Table 14-33: cmd Register Settings for ATA Payload Transfer[†]

Bits	Value	Comment
start_cmd	1	-

Bits	Value	Comment
ccs_expected	1	CCS is expected. Set to 1 for the RW_BLK command if interrupts are enabled in CE-ATA card device (the nIEN bit is set to 0 in the ATA control register)
read_ceata_device	0 or 1	Set to 1 for a RW_BLK or RW_REG read command
update_clk_regs_only	0	No clock parameters update command
card_num	0	-
send_initialization	0	No initialization sequence
stop_abort_cmd	0	-
send_auto_stop	0	-
transfer_mode	0	Block transfer mode. Byte count must be integer multiple of 4kB. Block size can be 512, 1k or 4k bytes
read_write	1 or 0	1 for write and 0 for read
data_expected	1	Data is expected
response_length	0	-
response_expect	1	-
cmd_index	Command index	Set this parameter to the command number. For example, set to 24 for SD/SDIO WRITE_BLOCK (CMD24) or 25 for WRITE_MULTIPLE_BLOCK (CMD25).
wait_prvdata_complete	1	0 for send command immediately 1 for send command after previous DTO interrupt
check_response_crc	1	0 for not checking response CRC 1 for checking response CRC

Table 14-34: blksiz Register Settings for ATA Payload Transfer[†]

Bits	Value	Comment
31:16	0	Reserved bits set to 0

Bits	Value	Comment
15:0 (block_size)	512, 1024 or 4096	MMC block size can be 512, 1024 or 4096 bytes as negotiated by host

Table 14-35: bytcnt Register Settings for ATA Payload Transfer

Bits	Value	Comment
31:0	$\langle n \rangle * \text{block\_size}$	Byte count must be an integer multiple of the block size. For ATA media access commands, byte count must be a multiple of 4 KB. $(\langle n \rangle * \text{block\_size} = \langle x \rangle * 4 \text{ KB})$, where $\langle n \rangle$ and $\langle x \rangle$ are integers

CE-ATA CCS

This section describes disabling the CCS, recovery after CCS timeout, and recovery after I/O read transmission delay (N_{ACIO}) timeout. [†]

Disabling the CCS

While waiting for the CCS for an outstanding RW_BLK command, the host can disable the CCS by sending a CCSD command:[†]

- Send a CCSD command—the controller sends the CCSD command to the CE-ATA card device if the send_ccsd bit is set to 1 in the ctrl register of the controller. This bit can be set only after a response is received for the RW_BLK command.[†]
- Send an internal stop command—send an internally-generated SD/SDIO STOP_TRANSMISSION (CMD12) command after sending the CCSD pattern. If the send_auto_stop_ccsd bit of the ctrl register is also set to 1 when the controller is set to send the CCSD pattern, the controller sends the internally-generated STOP command to the CMD pin. After sending the STOP command, the controller sets the acd bit in the rints register to 1.[†]

Recovery after CCS Timeout

If a timeout occurs while waiting for the CCS, the host needs to send the CCSD command followed by a STOP command to abort the pending ATA command. The host can set up the controller to send an internally-generated STOP command after sending the CCSD pattern:[†]

- Send CCSD command—set the send_ccsd bit in the ctrl register to 1.[†]
- Send external STOP command—terminate the data transfer between the CE-ATA card device and the controller. For more information about sending the STOP command, refer to *Transfer Stop and Abort Commands*.[†]
- Send internal STOP command—set the send_auto_stop_ccsd bit in the ctrl register to 1, which tells the controller to send the internally-generated STOP command. After sending the STOP command, the controller sets the acd bit in the rints register to 1. The send_auto_stop_ccsd bit must be set to 1 along with setting the send_ccsd bit.[†]

Related Information

[Transfer Stop and Abort Commands](#) on page 14-56

Refer to this section for more information about sending the STOP command.

Recovery after I/O Read Transmission Delay (N_{ACIO}) Timeout

If the I/O read transmission delay (N_{ACIO}) timeout occurs for the CE-ATA card device, perform one of the following steps to recover from the timeout:[†]

- If the CCS is expected from the CE-ATA card device (that is, the `ccs_expected` bit is set to 1 in the `cmd` register), follow the steps in *Recovery after CCS Timeout*.[†]
- If the CCS is not expected from the CE-ATA card device, perform the following steps: [†]
 1. Send an external STOP command. [†]
 2. Terminate the data transfer between the controller and CE-ATA card device. [†]

Related Information

[Recovery after CCS Timeout](#) on page 14-67

For more information about what steps to take if the CCS is expected from the CE-ATA card device.

Reduced ATA Command Set

It is necessary for the CE-ATA card device to support the reduced ATA command subset. This section describes the reduced command set.[†]

The IDENTIFY DEVICE Command

The IDENTIFY DEVICE command returns a 512-byte data structure to the host that describes device-specific information and capabilities. The host issues the IDENTIFY DEVICE command only if the MMC block size is set to 512 bytes. Any other MMC block size has indeterminate results.[†]

The host issues a `RW_REG` command for the ATA command, and the data is retrieved with the `RW_BLK` command.[†]

The host controller uses the following settings while sending a `RW_REG` command for the IDENTIFY DEVICE ATA command. The following list shows the primary bit settings:[†]

- `cmd` register setting: `data_expected` bit set to 0[†]
- `cmdarg` register settings: [†]
 - Bit [31] set to 0[†]
 - Bits [7:2] set to 128 [†]
 - All other bits set to 0[†]
- Task file settings: [†]
 - Command field of the ATA task file set to 0xEC[†]
 - Reserved fields of the task file set to 0[†]
- `bytcnt` register and `block_size` field of the `blksiz` register: set to 16[†]

The host controller uses the following settings for data retrieval (`RW_BLK` command):[†]

- cmd register settings:[†]
 - ccs_expected set to 1[†]
 - data_expected set to 1[†]
- cmdarg register settings: [†]
 - Bit [31] set to 0 (read operation) [†]
 - Bits [15:0] set to 1 (data unit count = 1)[†]
 - All other bits set to 0[†]
- bytcnt register and block_size field of the blksiz register: set to 512[†]

The READ DMA EXT Command

The READ DMA EXT command reads a number of logical blocks of data from the card device using the Data-In data transfer protocol. The host uses a RW_REG command to issue the ATA command and the RW_BLK command for the data transfer.[†]

The WRITE DMA EXT Command

The WRITE DMA EXT command writes a number of logical blocks of data to the card device using the Data-Out data transfer protocol. The host uses a RW_REG command to issue the ATA command and the RW_BLK command for the data transfer.[†]

The STANDBY IMMEDIATE Command

This ATA command causes the card device to immediately enter the most aggressive power management mode that still retains internal device context. No data transfer (RW_BLK) is expected for this command.[†]

For card devices that do not provide a power savings mode, the STANDBY IMMEDIATE command returns a successful status indication. The host issues a RW_REG command for the ATA command, and the status is retrieved with the SD/SDIO CMD39 or RW_REG command. Only the status field of the ATA task file contains the success status; there is no error status.[†]

The host controller uses the following settings while sending the RW_REG command for the STANDBY IMMEDIATE ATA command:[†]

- cmd register setting: data_expected bit set to 0[†]
- cmdarg register settings: [†]
 - Bit [31] set to 1 [†]
 - Bits [7:2] set to 4 [†]
 - All other bits set to 0 [†]
- Task file settings: [†]
 - Command field of the ATA task file set to 0xE0[†]
 - Reserved fields of the task file set to 0[†]
- bytcnt register and block_size field of the blksiz register: set to 16 [†]

The FLUSH CACHE EXT Command

For card devices that buffer/cache written data, the FLUSH CACHE EXT command ensures that buffered data is written to the card media. For cards that do not buffer written data, the FLUSH CACHE EXT command returns a success status. No data transfer (RW_BLK) is expected for this ATA command. [†]

The host issues a RW_REG command for the ATA command, and the status is retrieved with the SD/SDIO CMD39 or RW_REG command. There can be error status for this ATA command, in which case fields other than the status field of the ATA task file are valid.[†]

The host controller uses the following settings while sending the RW_REG command for the STANDBY IMMEDIATE ATA command:[†]

- cmd register setting: data_expected bit set to 0[†]
- cmdarg register settings:[†]
 - Bit [31] set to 1[†]
 - Bits [7:2] set to 4[†]
 - All other bits set to 0[†]
- Task file settings:[†]
 - Command field of the ATA task file set to 0xEA[†]
 - Reserved fields of the task file set to 0[†]
- bytcnt register and block_size field of the blksiz register: set to 16[†]

Card Read Threshold

When an application needs to perform a single or multiple block read command, the application must set the cardthrctl register with the appropriate card read threshold size in the card read threshold field (cardrdthreshold) and set the cardrdthren bit to 1. This additional information specified in the controller ensures that the controller sends a read command only if there is space equal to the card read threshold available in the RX FIFO buffer. This in turn ensures that the card clock is not stopped in the middle a block of data being transmitted from the card. Set the card read threshold to the block size of the transfer to guarantee there is a minimum of one block size of space in the RX FIFO buffer before the controller enables the card clock.[†]

The card read threshold is required when the round trip delay is greater than half of sdmmc_clk_divided.[†]

Table 14-36: Card Read Threshold Guidelines[†]

Bus Speed Modes	Round Trip Delay (Delay_R) ⁽⁴⁸⁾	Is Stopping of Card Clock Allowed?	Card Read Threshold Required?
SDR25	$\text{Delay\_R} > 0.5 * (\text{sdmmc\_clk}/4)$	No	Yes
	$\text{Delay\_R} < 0.5 * (\text{sdmmc\_clk}/4)$	Yes	No
SDR12	$\text{Delay\_R} > 0.5 * (\text{sdmmc\_clk}/4)$	No	Yes
	$\text{Delay\_R} < 0.5 * (\text{sdmmc\_clk}/4)$	Yes	No

Related Information

[Intel Arria V Device Datasheet](#)

⁽⁴⁸⁾ $\text{Delay\_R} = \text{Delay\_O} + \text{tODLY} + \text{Delay\_I}$ [†]

Where:[†]

$\text{Delay\_O} = \text{sdmmc\_clk to sdmmc\_clk\_out delay (including I/O pin delay)}$ [†]

$\text{Delay\_I} = \text{Input I/O pin delay + routing delay to the input register}$ [†]

$\text{tODLY} = \text{sdmmc\_clk\_out to card output delay (varies across card manufactures and speed modes)}$ [†]

For the delay numbers needed for above calculation, refer to Arria V Datasheet.[†]

Recommended Usage Guidelines for Card Read Threshold

1. The `cardthrc1` register must be set before setting the `cmd` register for a data read command.[†]
2. The `cardthrc1` register must not be set while a data transfer command is in progress.[†]
3. The `cardrdthreshold` field of the `cardthrc1` register must be set to at the least the block size of a single or multiblock transfer. A `cardrdthreshold` field setting greater than or equal to the block size of the read transfer ensures that the card clock does not stop in the middle of a block of data.[†]
4. If the round trip delay is greater than half of the card clock period, card read threshold must be enabled and the card threshold must be set as per guideline 3 to guarantee that the card clock does not stop in the middle of a block of data.[†]
5. If the `cardrdthreshold` field is set to less than the block size of the transfer, the host must ensure that the receive FIFO buffer never overflows during the read transfer. Overflow can cause the card clock from the controller to stop. The controller is not able to guarantee that the card clock does not stop during a read transfer.[†]

Note: If the `cardrdthreshold` field of the `cardthrc1` register, and the `rx_wmark` and `dw_dma_multiple_transaction_size` fields of the `fifoth` register are set incorrectly, the card clock might stop indefinitely, with no interrupts generated by the controller.[†]

Card Read Threshold Programming Sequence

Most cards, such as SDHC or SDXC, support block sizes that are either specified in the card or are fixed to 512 bytes. For SDIO, MMC, and standard capacity SD cards that support partial block read (`READ_BL_PARTIAL` set to 1 in the CSD register of the card device), the block size is variable and can be chosen by the application.[†]

To use the card read threshold feature effectively and to guarantee that the card clock does not stop because of a FIFO Full condition in the middle of a block of data being read from the card, follow these steps:[†]

1. Choose a block size that is a multiple of four bytes.[†]
2. Enable card read threshold feature. The card read threshold can be enabled only if the block size for the given transfer is less than or equal to the total depth of the FIFO buffer:[†]
$$(\text{block size} / 4) \leq 1024^{\dagger}$$
3. Choose the card read threshold value: [†]
 - If $(\text{block size} / 4) \geq 512$, choose `cardrdthreshold` such that:[†]
 - $\text{cardrdthreshold} \leq (\text{block size} / 4)$ in bytes[†]
 - If $(\text{block size} / 4) < 512$, choose `cardrdthreshold` such that:[†]
 - $\text{cardrdthreshold} = (\text{block size} / 4)$ in bytes[†]
4. Set the `dw_dma_multiple_transaction_size` field in the `fifoth` register to the number of transfers that make up a DMA transaction. For example, `size = 1` means 4 bytes are moved. The possible values for the size are 1, 4, 8, 16, 32, 64, 128, and 256 transfers. Select the size so that the value $(\text{block size} / 4)$ is evenly divided by the size.[†]
5. Set the `rx_wmark` field in the `fifoth` register to the `size - 1`.[†]

For example, if your block size is 512 bytes, legal values of `dw_dma_multiple_transaction_size` and `rx_wmark` are listed in the following table.

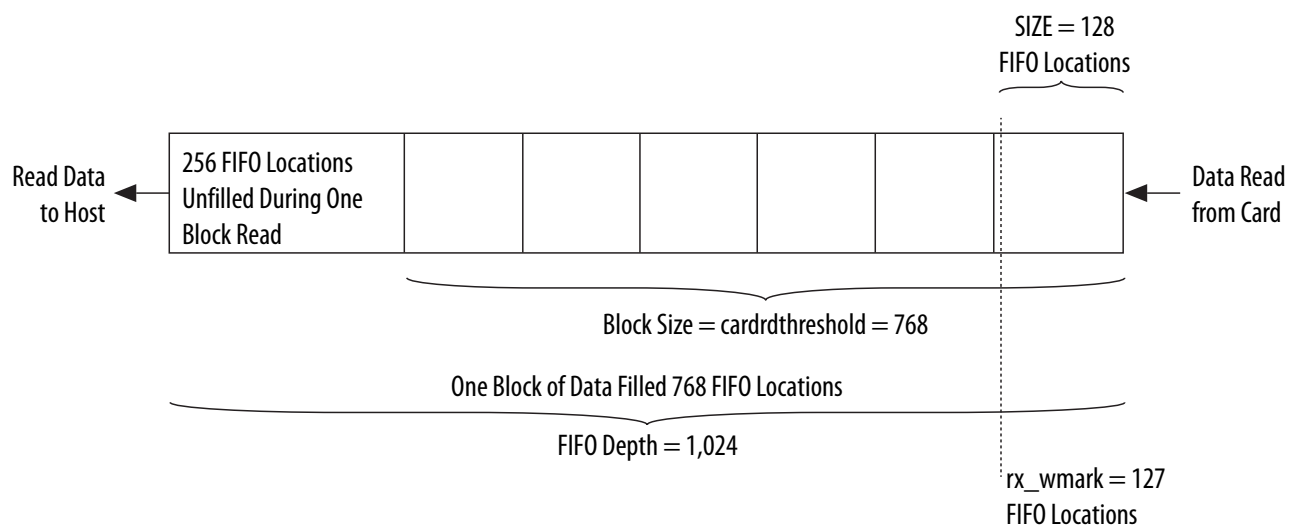
Table 14-37: Legal Values of `dw_dma_multiple_transaction_size` and `rx_wmark` for Block Size = 512[†]

Block Size	<code>dw_dma_multiple_transaction_size</code>	<code>rx_wmark</code>
512	1	0
512	4	3
512	8	7
512	16	15
512	32	31
512	64	63
512	128	127

Card Read Threshold Programming Examples

This section shows examples of how to program the card read threshold.[†]

- Choose a block size that is a multiple of 4 (the number of bytes per FIFO location), and less than 4096 (1024 FIFO locations). For example, a block size of 3072 bytes is legal, because $3072 / 4 = 768$ FIFO locations.[†]
- For DMA mode, choose the size so that block size is a multiple of the size. For example size = 128, where $\text{block size} \% \text{size} = 0$.[†]
- Set the `rx_wmark` field = size - 1. For example, the `rx_wmark` field = $128 - 1 = 127$.[†]
- Because $\text{block size} > \frac{1}{2} \text{FifoDepth}$, set the `cardrdthreshold` field to the block size. For example, the `cardrdthreshold` field = 3072 bytes.[†]

Figure 14-16: FIFO Buffer content when Card Read Threshold is set to 768[†]

Interrupt and Error Handling

This section describes how to use interrupts to handle errors. On power-on or reset, interrupts are disabled (the `int_enable` bit in the `ctrl` register is set to 0), and all the interrupts are masked (the `intmask` register default is 0). The controller error handling includes the following types of errors:

- Response and data timeout errors—For response time-outs, the host software can retry the command. For data time-outs, the controller has not received the data start bit from the card, so software can either retry the whole data transfer again or retry from a specified block onwards. By reading the contents of the `tcrcnt` register later, the software can decide how many bytes remain to be copied (read).[†]
- Response errors—Set to 1 when an error is received during response reception. If the response received is invalid, the software can retry the command.[†]
- Data errors—Set to 1 when a data receive error occurs. Examples of data receive errors:[†]
 - Data CRC[†]
 - Start bit not found[†]
 - End bit not found[†]

These errors can be occur on any block. On receipt of an error, the software can issue an SD/SDIO STOP or SEND_IF_COND command, and retry the command for either the whole data or partial data.[†]

- Hardware locked error—Set to 1 when the controller cannot load a command issued by software. When software sets the `start_cmd` bit in the `cmd` register to 1, the controller tries to load the command. If the command buffer already contains a command, this error is raised, and the new command is discarded, requiring the software to reload the command.[†]
- FIFO buffer underrun/overflow error—If the FIFO buffer is full and software tries to write data to the FIFO buffer, an overflow error is set. Conversely, if the FIFO buffer is empty and the software tries to read data from the FIFO buffer, an underrun error is set. Before reading or writing data in the FIFO buffer, the software must read the FIFO buffer empty bit (`fifo_empty`) or FIFO buffer full bit (`fifo_full`) in the status register.[†]
- Data starvation by host timeout—This condition occurs when software does not service the FIFO buffer fast enough to keep up with the controller. Under this condition and when a read transfer is in process, the software must read data from the FIFO buffer, which creates space for further data reception. When a transmit operation is in process, the software must write data to fill the FIFO buffer so that the controller can write the data to the card.[†]
- CE-ATA errors[†]
- CRC error on command—If a CRC error is detected for a command, the CE-ATA card device does not send a response, and a response timeout is expected from the controller. The ATA layer is notified that an MMC transport layer error occurred.

- CRC error on command—If a CRC error is detected for a command, the CE-ATA card device does not send a response, and a response timeout is expected from the controller. The ATA layer is notified that an MMC transport layer error occurred.[†]
- Write operation—Any MMC transport layer error known to the card device causes an outstanding ATA command to be terminated. The ERR bits are set in the ATA status registers and the appropriate error code is sent to the Error Register (Error) on the ATA card device.[†]

If the device interrupt bit of the CE-ATA card (the nIEN bit in the ATA control register) is set to 0, the CCS is sent to the host.[†]

If the device interrupt bit is set to 1, the card device completes the entire data unit count if the host controller does not abort the ongoing transfer.[†]

Note: During a multiple-block data transfer, if a negative CRC status is received from the card device, the data path signals a data CRC error to the BIU by setting the `dcrc` bit in the `rintsts` register to 1. It then continues further data transmission until all the bytes are transmitted.[†]

- Read operation—If MMC transport layer errors are detected by the host controller, the host completes the ATA command with an error status. The host controller can issue a CCSD command followed by a STOP_TRANSMISSION (CMD12) command to abort the read transfer. The host can also transfer the entire data unit count bytes without aborting the data transfer.[†]

Bootling Operation for eMMC and MMC

This section describes how to set up the controller for eMMC and MMC boot operation.

Note: The BootROM and initial software do not use the boot partitions that are in the MMC card. This means that there is no boot partition support of the SD/MMC controller.

Boot Operation by Holding Down the CMD Line

The controller can boot from MMC4.3, MMC4.4, and MMC4.41 cards by holding down the CMD line.

For information about this boot method, refer to the following specifications, available on the JEDEC website:

- JEDEC Standard No. 84-A441
- JEDEC Standard No. 84-A44
- JEDEC Standard No. JESD84-A43

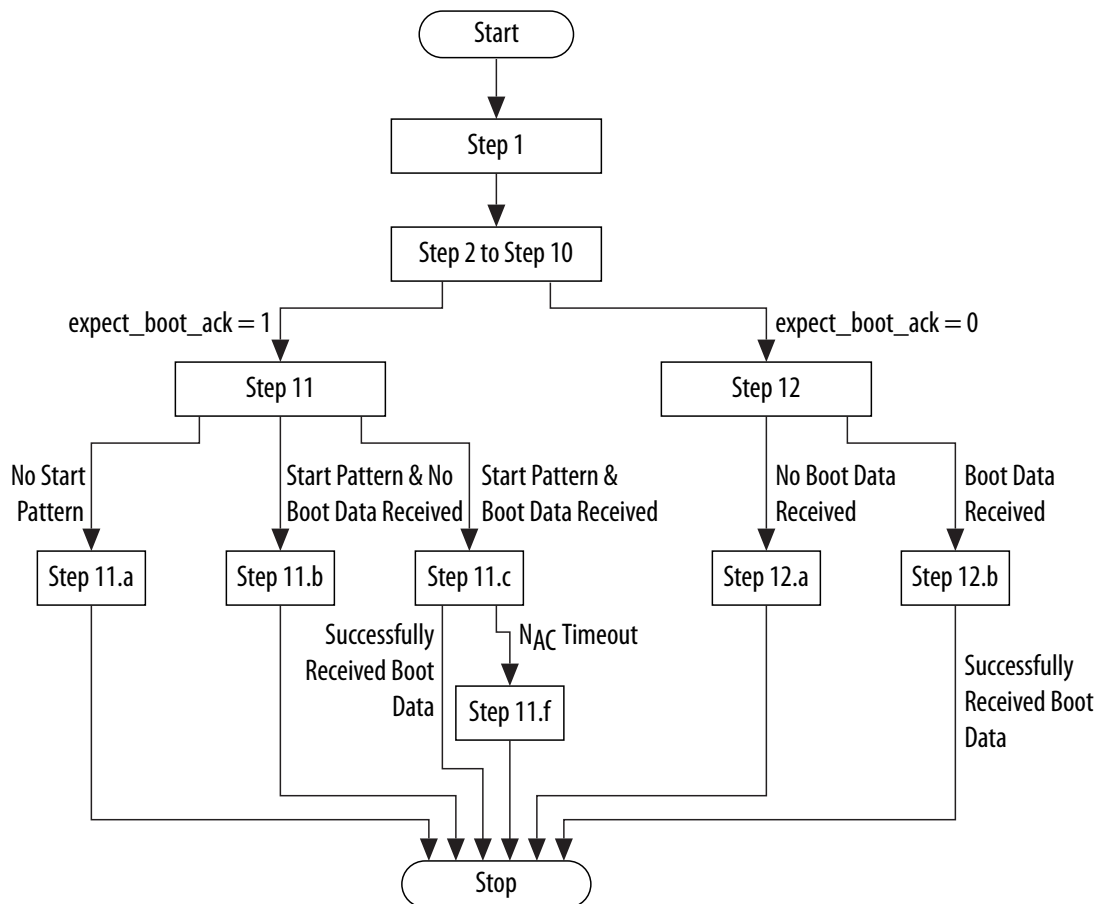
Related Information

[JEDEC Global Standards of the Microelectronics Industry](#)

For more information about this boot method, refer to the following JEDEC Standards available on the JEDEC website: No. 84-A441, No. 84-A44, and No. JESD84-A43.

Boot Operation for eMMC Card Device

The following figure illustrates the steps to perform the boot process for eMMC card devices. The detailed steps are described following the flow chart.

Figure 14-17: Flow for eMMC Boot Operation[†]

1. The software driver performs the following checks: [†]

- If the eMMC card device supports boot operation (the BOOT_PARTITION_ENABLE bit is set to 1 in the EXT_CSD register of the eMMC card).[†]
- The BOOT_SIZE_MULT and BOOT_BUS_WIDTH values in the EXT_CSD register, to be used during the boot process.[†]

2. The software sets the following bits: [†]

- Sets masks for interrupts, by setting the appropriate bits to 0 in the intmask register.[†]
- Sets the global int_enable bit of the ctrl register to 1. Other bits in the ctrl register must be set to 0. [†]

Note: Intel recommends that you write 0xFFFFFFFF to the rintsts and idsts registers to clear any pending interrupts before setting the int_enable bit. For internal DMA controller mode, the software driver needs to unmask all the relevant fields in the idinten register.[†]

3. If the software driver needs to use the internal DMA controller to transfer the boot data received, it must perform the following steps: [†]

- Set up the descriptors as described in *Internal DMA Controller Transmission Sequences* and *Internal DMA Controller Reception Sequences*.[†]
 - Set the use_internal_dmac bit of the ctrl register to 1.[†]
4. Set the card device frequency to 400 kHz using the clkdiv registers. For more information, refer to Clock Setup.[†]
 5. Set the data_timeout field of the tmount register equal to the card device total access time, N_{AC} .[†]
 6. Set the blksiz register to 0x200 (512 bytes).[†]
 7. Set the bytcnt register to a multiple of 128 KB, as indicated by the BOOT_SIZE_MULT value in the card device.[†]
 8. Set the rx_wmark field in the fifoth register. Typically, the threshold value can be set to 512, which is half the FIFO buffer depth.[†]
 9. Set the following fields in the cmd register:[†]
 - Initiate the command by setting start_cmd = 1[†]
 - Enable boot (enable_boot) = 1[†]
 - Expect boot acknowledge (expect_boot_ack):[†]
 - If a start-acknowledge pattern is expected from the card device, set expect_boot_ack to 1.[†]
 - If a start-acknowledge pattern is not expected from the card device, set expect_boot_ack to 0.[†]
 - Card number (card_number) = 0[†]
 - data_expected = 1[†]
 - Reset the remainder of cmd register bits to 0[†]
 10. If no start-acknowledge pattern is expected from the card device (expect_boot_ack set to 0) proceed to [step 12](#).[†]
 11. This step handles the case where a start-acknowledge pattern is expected (expect_boot_ack was set to 1 in [step 9](#)).[†]
 - a. If the Boot ACK Received interrupt is not received from the controller within 50 ms of initiating the command ([step 9](#)), the software driver must set the following cmd register fields:[†]
 - start_cmd = 1[†]
 - Disable boot (disable_boot) = 1[†]
 - card_number = 0[†]
 - All other fields = 0[†]

The controller generates a Command Done interrupt after deasserting the CMD pin of the card interface.[†]

If internal DMA controller mode is used for the boot process, the controller performs the following steps after the Boot ACK Received timeout:[†]

- The DMA descriptor is closed.[†]
 - The ces bit in the idsts register is set, indicating the Boot ACK Received timeout.[†]
 - The ri bit of the idsts register is not set.[†]
- b. If the Boot ACK Received interrupt is received, the software driver must clear this interrupt by writing 1 to the ces bit in the idsts register.[†]

Within 0.95 seconds of the Boot ACK Received interrupt, the Boot Data Start interrupt must be received from the controller. If this does not occur, the software driver must write the following cmd register fields:[†]

- start_cmd = 1[†]
- disable_boot = 1[†]
- card_number = 0[†]
- All other fields = 0[†]

The controller generates a Command Done interrupt after deasserting the CMD pin of the card interface.[†]

If internal DMA controller mode is used for the boot process, the controller performs the following steps after the Boot ACK Received timeout:[†]

- The DMA descriptor is closed[†]
 - The `ces` bit in the `idsts` register is set, indicating Boot Data Start timeout[†]
 - The `ri` bit of the `idsts` register is not set[†]
- c. If the Boot Data Start interrupt is received, it indicates that the boot data is being received from the card device. When the DMA engine is not in internal DMA controller mode, the software driver can then initiate a data read from the controller based on the `rxdr` interrupt bit in the `rintsts` register.[†]

In internal DMA controller mode, the DMA engine starts transferring the data from the FIFO buffer to the system memory as soon as the level set in the `rx_wmark` field of the `fifoth` register is reached.[†]

At the end of a successful boot data transfer from the card, the following interrupts are generated:[†]

- The `cmd` bit and `dto` bit in the `rintsts` register[†]
 - The `ri` bit in the `idsts` register, in internal DMA controller mode only[†]
- d. If an error occurs in the boot ACK pattern (0b010) or an EBE occurs:[†]
- The controller automatically aborts the boot process by pulling the CMD line high[†]
 - The controller generates a Command Done interrupt[†]
 - The controller does not generate a Boot ACK Received interrupt[†]
 - The application aborts the boot transfer[†]
- e. In internal DMA controller mode:[†]
- If the software driver creates more descriptors than required by the received boot data, the extra descriptors are not closed by the controller. Software cannot reuse the descriptors until they are closed.[†]
 - If the software driver creates fewer descriptors than required by the received boot data, the controller generates a Descriptor Unavailable interrupt and does not transfer any further data to system memory.[†]
- f. If N_{AC} is violated between data block transfers, the DRTO interrupt is asserted. In addition, if there is an error associated with the start or end bit, the SBE or EBE interrupt is also generated.[†]

The boot operation for eMMC card devices is complete. Do not execute the remaining ([step 12](#)).[†]

12. This step handles the case where no start-acknowledge pattern is expected (`expect_boot_ack` was set to 0 in [step 9](#)).[†]

- a. If the Boot Data Start interrupt is not received from the controller within 1 second of initiating the command ([step 9](#)), the software driver must write the `cmd` register with the following fields:[†]
- `start_cmd` = 1[†]
 - `disable_boot` = 1[†]
 - `card_number` = 0[†]
 - All other fields = 0[†]

The controller generates a Command Done interrupt after deasserting the CMD line of the card. In internal DMA controller mode, the descriptor is closed and the `ces` bit in the `idsts` register is set to 1, indicating a Boot Data Start timeout.[†]

- b. If a Boot Data Start interrupt is received, it indicates that the boot data is being received from the card device. When the DMA engine is not in internal DMA controller mode, the software driver

can then initiate a data read from the controller based on the `rxdr` interrupt bit in the `rintsts` register.[†]

In internal DMA controller mode, the DMA engine starts transferring the data from the FIFO buffer to the system memory as soon as the level specified in the `rx_wmark` field of the `fifoth` register is reached.[†]

At the end of a successful boot data transfer from the card, the following interrupts are generated:[†]

- The `cmd` bit and `dto` bit in the `rintsts` register[†]
- The `ri` bit in the `idsts` register, in internal DMA controller mode only[†]
- c. In internal DMA controller mode:[†]
 - If the software driver creates more descriptors than required by the received boot data, the extra descriptors are not closed by the controller.[†]
 - If the software driver creates fewer descriptors than required by the received boot data, the controller generates a Descriptor Unavailable interrupt and does not transfer any further data to system memory.[†]

The boot operation for eMMC card devices is complete.[†]

Related Information

- [Clock Setup](#) on page 14-46
Refer to this section for information on how to set the card device frequency.
- [Internal DMA Controller Transmission Sequences](#) on page 14-59
Refer to this section for information about the Internal DMA Controller Transmission Sequences.
- [Internal DMA Controller Reception Sequences](#) on page 14-59
Refer to this section for information about the Internal DMA Controller Reception Sequences.

Boot Operation for Removable MMC4.3, MMC4.4 and MMC4.41 Cards

Removable MMC4.3, MMC4.4, and MMC4.41 Differences

Removable MMC4.3, MMC4.4, and MMC4.41 cards differ with respect to eMMC in that the controller is not aware whether these cards support the boot mode of operation when plugged in. Thus, the controller must:[†]

1. Discover these cards as it would discover MMC4.0/4.1/4.2 cards for the first time[†]
2. Know the card characteristics[†]
3. Decide whether to perform a boot operation or not[†]

Bootting Removable MMC4.3, MMC4.4 and MMC4.41 Cards

For removable MMC4.3, MMC4.4 and MMC4.41 cards, the software driver must perform the following steps:[†]

1. Discover the card as described in *Enumerated Card Stack*.[†]
2. Read the `EXT_CSD` register of the card and examine the following fields:[†]
 - `BOOT_PARTITION_ENABLE`[†]
 - `BOOT_SIZE_MULT`[†]
 - `BOOT_INFO`[†]
3. If necessary, the software can manipulate the boot information in the card.[†]

Note: For more information, refer to “Access to Boot Partition” in the following specifications available on the JEDEC website:

- JEDEC Standard No. 84-A441
 - JEDEC Standard No. 84-A44
 - JEDEC Standard No. JESD84-A43
4. If the host processor needs to perform a boot operation at the next power-up cycle, it can manipulate the EXT_CSD register contents by using a SWITCH_FUNC command.[†]
 5. After this step, the software driver must power down the card by writing to the pwren register.[†]
 6. From here on, use the same steps as in *Alternative Boot Operation for eMMC Card Devices*.[†]

Related Information

- [Enumerated Card Stack](#) on page 14-43
Refer to this section for more information on discovering removable MMC cards.
- <http://www.jedec.org>
- [Alternative Boot Operation for eMMC Card Devices](#) on page 14-79
Refer to this section for information about alternative boot operation steps.

Alternative Boot Operation

The alternative boot operation differs from the previous boot operation in that software uses the SD/SDIO GO_IDLE_STATE command to boot the card, rather than holding down the CMD line of the card. The alternative boot operation can be performed only if bit 0 in the BOOT_INFO register is set to 1. BOOT_INFO is located at offset 228 in the EXT_CSD registers.[†]

For detailed information about alternative boot operation, refer to the following specifications available on the JEDEC website:

- JEDEC Standard No. 84-A441
- JEDEC Standard No. 84-A44
- JEDEC Standard No. JESD84-A43

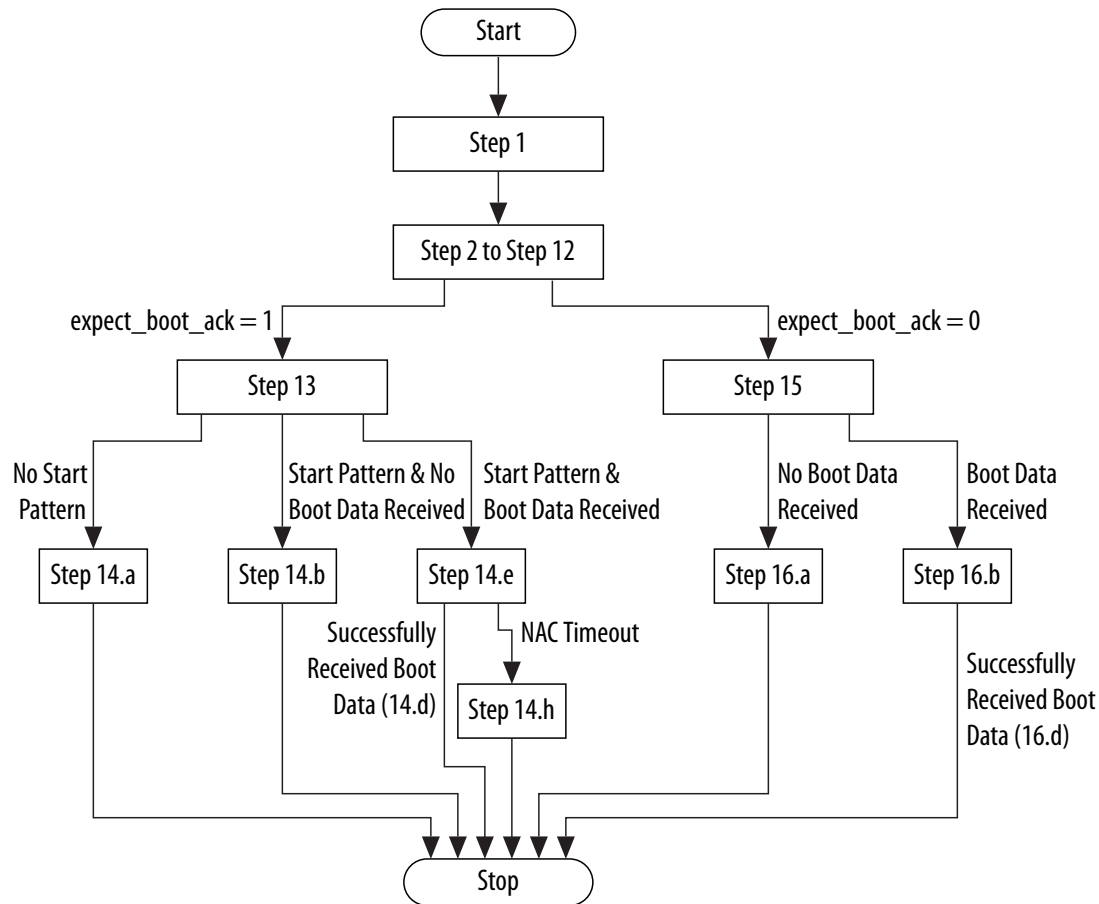
Related Information

[JEDEC Global Standards of the Microelectronics Industry](#)

For more information about alternative boot operation, refer to the following JEDEC Standards available on the JEDEC website: No. 84-A441, No. 84-A44, and No. JESD84-A43.

Alternative Boot Operation for eMMC Card Devices

The following figure illustrates the sequence of steps required to perform the alternative boot operation for eMMC card devices. The detailed steps are described following the flow chart.

Figure 14-18: Flow for eMMC Alternative Boot Operation[†]

1. The software driver checks:[†]

- If the eMMC card device supports alternative boot operation (the BOOT_INFO bit is set to 1 in the eMMC card).[†]
- The BOOT_SIZE_MULT and BOOT_BUS_WIDTH values in the card device to use during the boot process.[†]

2. The software sets the following bits:[†]

- Sets masks for interrupts by resetting the appropriate bits to 0 in the intmask register.[†]
- Sets the int_enable bit of the ctrl register to 1. Other bits in the ctrl register must be set to 0.[†]

Note: Intel recommends writing 0xFFFFFFFF to the rintsts register and idsts register to clear any pending interrupts before setting the int_enable bit. For internal DMA controller mode, the software driver needs to unmask all the relevant fields in the idinten register.[†]

3. If the software driver needs to use the internal DMA controller to transfer the boot data received, it must perform the following actions:[†]

- Set up the descriptors as described in *Internal DMA Controller Transmission Sequences* and *Internal DMA Controller Reception Sequences*.[†]
- Set the use internal DMAC bit (use_internal_dmac) of the ctrl register to 1.[†]

4. Set the card device frequency to 400 kHz using the clkdiv registers. For more information, refer to *Clock Setup*. Ensure that the card clock is running.[†]

5. Wait for a time that ensures that at least 74 card clock cycles have occurred on the card interface.[†]
6. Set the `data_timeout` field of the `tmout` register equal to the card device total access time, N_{AC} .[†]
7. Set the `blksiz` register to 0x200 (512 bytes).[†]
8. Set the `bytcnt` register to multiples of 128K bytes, as indicated by the `BOOT_SIZE_MULT` value in the card device.[†]
9. Set the `rx_wmark` field in the `fifoth` register. Typically, the threshold value can be set to 512, which is half the FIFO buffer depth.[†]
10. Set the `cmdarg` register to 0xFFFFFFFF.[†]
11. Initiate the command, by setting the `cmd` register with the following fields:[†]
 - `start_cmd` = 1[†]
 - `enable_boot` = 1[†]
 - `expect_boot_ack`:[†]
 - If a start-acknowledge pattern is expected from the card device, set `expect_boot_ack` to 1.[†]
 - If a start-acknowledge pattern is not expected from the card device, set `expect_boot_ack` to 0.[†]
 - `card_number` = 0[†]
 - `data_expected` = 1[†]
 - `cmd_index` = 0[†]
 - Set the remainder of `cmd` register bits to 0.[†]
12. If no start-acknowledge pattern is expected from the card device (`expect_boot_ack` set to 0) jump to [step 15](#).[†]
13. Wait for the Command Done interrupt.[†]
14. This step handles the case where a start-acknowledge pattern is expected (`expect_boot_ack` was set to 1 in [step 11](#)).[†]
 - a. If the Boot ACK Received interrupt is not received from the controller within 50 ms of initiating the command ([step 11](#)), the start pattern was not received. The software driver must discontinue the boot process and start with normal discovery.[†]

If internal DMA controller mode is used for the boot process, the controller performs the following steps after the Boot ACK Received timeout:[†]

 - The DMA descriptor is closed.[†]
 - The `ces` bit in the `idsts` register is set to 1, indicating the Boot ACK Received timeout.[†]
 - The `ri` bit of the `idsts` register is not set.[†]
 - b. If the Boot ACK Received interrupt is received, the software driver must clear this interrupt by writing 1 to it.[†]

Within 0.95 seconds of the Boot ACK Received interrupt, the Boot Data Start interrupt must be received from the controller. If this does not occur, the software driver must discontinue the boot process and start with normal discovery.[†]

If internal DMA controller mode is used for the boot process, the controller performs the following steps after the Boot ACK Received timeout:[†]

 - The DMA descriptor is closed.[†]
 - The `ces` bit in the `idsts` register is set to 1, indicating Boot Data Start timeout.[†]
 - The `ri` bit of the `idsts` register is not set.[†]
 - c. If the Boot Data Start interrupt is received, it indicates that the boot data is being received from the card device. When the DMA engine is not in internal DMA controller mode, the software driver can then initiate a data read from the controller based on the `rxdr` interrupt bit in the `rintsts` register.[†]



In internal DMA controller mode, the DMA engine starts transferring the data from the FIFO buffer to the system memory as soon as the level specified in the `rx_wmark` field of the `fifoth` register is reached.[†]

- d. The software driver must terminate the boot process by instructing the controller to send the SD/SDIO `GO_IDLE_STATE` command:[†]
 - Reset the `cmdarg` register to 0.[†]
 - Set the `start_cmd` bit of the `cmd` register to 1, and all other bits to 0.[†]
- e. At the end of a successful boot data transfer from the card, the following interrupts are generated:[†]
 - The `cmd` bit and `dto` bit in the `rintsts` register[†]
 - The `ri` bit in the `idsts` register, in internal DMA controller mode only[†]
- f. If an error occurs in the boot ACK pattern (0b010) or an EBE occurs:[†]
 - The controller does not generate a Boot ACK Received interrupt.[†]
 - The controller detects Boot Data Start and generates a Boot Data Start interrupt.[†]
 - The controller continues to receive boot data.[†]
 - The application must abort the boot process after receiving a Boot Data Start interrupt.[†]
- g. In internal DMA controller mode:[†]
 - If the software driver creates more descriptors than required by the received boot data, the extra descriptors are not closed by the controller.[†]
 - If the software driver creates fewer descriptors than required by the received boot data, the controller generates a Descriptor Unavailable interrupt and does not transfer any further data to system memory.[†]
- h. If N_{AC} is violated between data block transfers, a DRTO interrupt is asserted. Apart from this, if there is an error associated with the start or end bit, the SBE or EBE interrupt is also generated.[†]

The alternative boot operation for eMMC card devices is complete. Do not execute the remaining steps (15 and 16).[†]

15. Wait for the Command Done interrupt.[†]

16. This step handles the case where a start-acknowledge pattern is not expected (`expect_boot_ack` was set to 0 in [step 11](#)).[†]

- a. If the Boot Data Start interrupt is not received from the controller within 1 second of initiating the command ([step 11](#)), the software driver must discontinue the boot process and start with normal discovery.[†] In internal DMA controller mode:[†]
 - The DMA descriptor is closed.[†]
 - The `ces` bit in the `idsts` register is set to 1, indicating Boot Data Start timeout.[†]
 - The `ri` bit of the `idsts` register is not set.[†]
- b. If a Boot Data Start interrupt is received, the boot data is being received from the card device. When the DMA engine is not in internal DMA controller mode, the software driver can then initiate a data read from the controller based on the `rxdr` interrupt bit in the `rintsts` register.[†]

In internal DMA controller mode, the DMA engine starts transferring the data from the FIFO buffer to the system memory as soon as the level specified in the `rx_wmark` field of the `fifoth` register is reached.[†]

- c. The software driver must terminate the boot process by instructing the controller to send the SD/SDIO `GO_IDLE_STATE` (`CMD0`) command:[†]
 - Reset the `cmdarg` register to 0.[†]
 - Set the `start_cmd` bit in the `cmd` register to 1, and all other bits to 0.[†]
- d. At the end of a successful boot data transfer from the card, the following interrupts are generated:[†]

- The cmd bit and dto bit in the rintsts register[†]
- The ri bit in the idsts register, in internal DMA controller mode only[†]
- e. In internal DMA controller mode: [†]
 - If the software driver creates more descriptors than required by the received boot data, the extra descriptors are not closed by the controller. [†]
 - If the software driver creates fewer descriptors than required by the received boot data, the controller generates a Descriptor Unavailable interrupt and does not transfer any further data to system memory. [†]

The alternative boot operation for eMMC card devices is complete. [†]

Related Information

- **Clock Setup** on page 14-46
Refer to this section for information on how to set the card device frequency.
- **Internal DMA Controller Transmission Sequences** on page 14-59
Refer to this section for information about the Internal DMA Controller Transmission Sequences.
- **Internal DMA Controller Reception Sequences** on page 14-59
Refer to this section for information about the Internal DMA Controller Reception Sequences.

Alternative Boot Operation for MMC4.3 Cards

Removable MMC4.3 Boot Mode Support

Removable MMC4.3 cards differ with respect to eMMC in that the controller is not aware whether these cards support the boot mode of operation. Thus, the controller must: [†]

1. Discover these cards as it would discover MMC4.0/4.1/4.2 cards for the first time [†]
2. Know the card characteristics [†]
3. Decide whether to perform a boot operation or not[†]

Discovering Removable MMC4.3 Boot Mode Support

For removable MMC4.3 cards, the software driver must perform the following steps: [†]

1. Discover the card as described in *Enumerated Card Stack*.[†]
2. Read the MMC card device's EXT_CSD registers and examine the following fields: [†]
 - BOOT_PARTITION_ENABLE [†]
 - BOOT_SIZE_MULT [†]
 - BOOT_INFO [†]

Note: For more information, refer to "Access to Boot Partition" in JEDEC Standard No. JESD84-A43, available on the JEDEC website. [†]

3. If the host processor needs to perform a boot operation at the next power-up cycle, it can manipulate the contents of the EXT_CSD registers in the MMC card device, by using a SWITCH_FUNC command. [†]
4. After this step, the software driver must power down the card by writing to the pwren register. [†]
5. From here on, use the same steps as in *Alternative Boot Operation for eMMC Card Devices*. [†]

Note: Ignore the EBE if it is generated during an abort scenario.

If a boot acknowledge error occurs, the boot acknowledge received interrupt times out. [†]

In internal DMA controller mode, the application needs to depend on the descriptor close interrupt instead of the data done interrupt. [†]

Related Information

- [Enumerated Card Stack](#) on page 14-43
Refer to this section for more information on discovering removable MMC cards.
- [JEDEC Global Standards of the Microelectronics Industry](#)
For more information, refer to "Access to Boot Partition" in JEDEC Standard No. JESD84-A43, available on the JEDEC website.
- [Alternative Boot Operation for eMMC Card Devices](#) on page 14-79
Refer to this section for information about alternative boot operation steps.

SD/MMC Controller Address Map and Register Definitions

The address map and register definitions for the SD/MMC Controller consists of the following region:

- SD/MMC Module

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) provides a quad serial peripheral interface (SPI) flash controller for access to serial NOR flash devices. The quad SPI flash controller supports standard SPI flash devices as well as high-performance dual and quad SPI flash devices. The quad SPI flash controller is based on Cadence Quad SPI Flash Controller (QSPI_FLASH_CTRL).

Features of the Quad SPI Flash Controller

The quad SPI flash controller supports the following features:

- Single, dual, and quad I/O commands
- Device frequencies up to 108 MHz⁰
- Direct access and indirect access modes
- External direct memory access (DMA) controller support for indirect transfers
- Configurable clock polarity and phase
- Supports 108 MHz (MAX) clock frequency for all protocols in single transfer rate (STR) mode.⁽⁴⁹⁾
- Programmable write-protected regions
- Programmable delays between transactions
- Programmable device sizes
- Read data capture tuning
- Local buffering with error correction code (ECC) logic for indirect transfers
- Up to four devices
- Supports the Micron N25Q512A (512 Mb, 108 MHz) and Micron N25Q00AA (1024 Mb, 108 MHz) Quad SPI flash memories that are verified working with the HPS
- eXecute-In-Place (XIP) mode

Related Information

Supported Flash Devices for Cyclone V and Arria V SoC

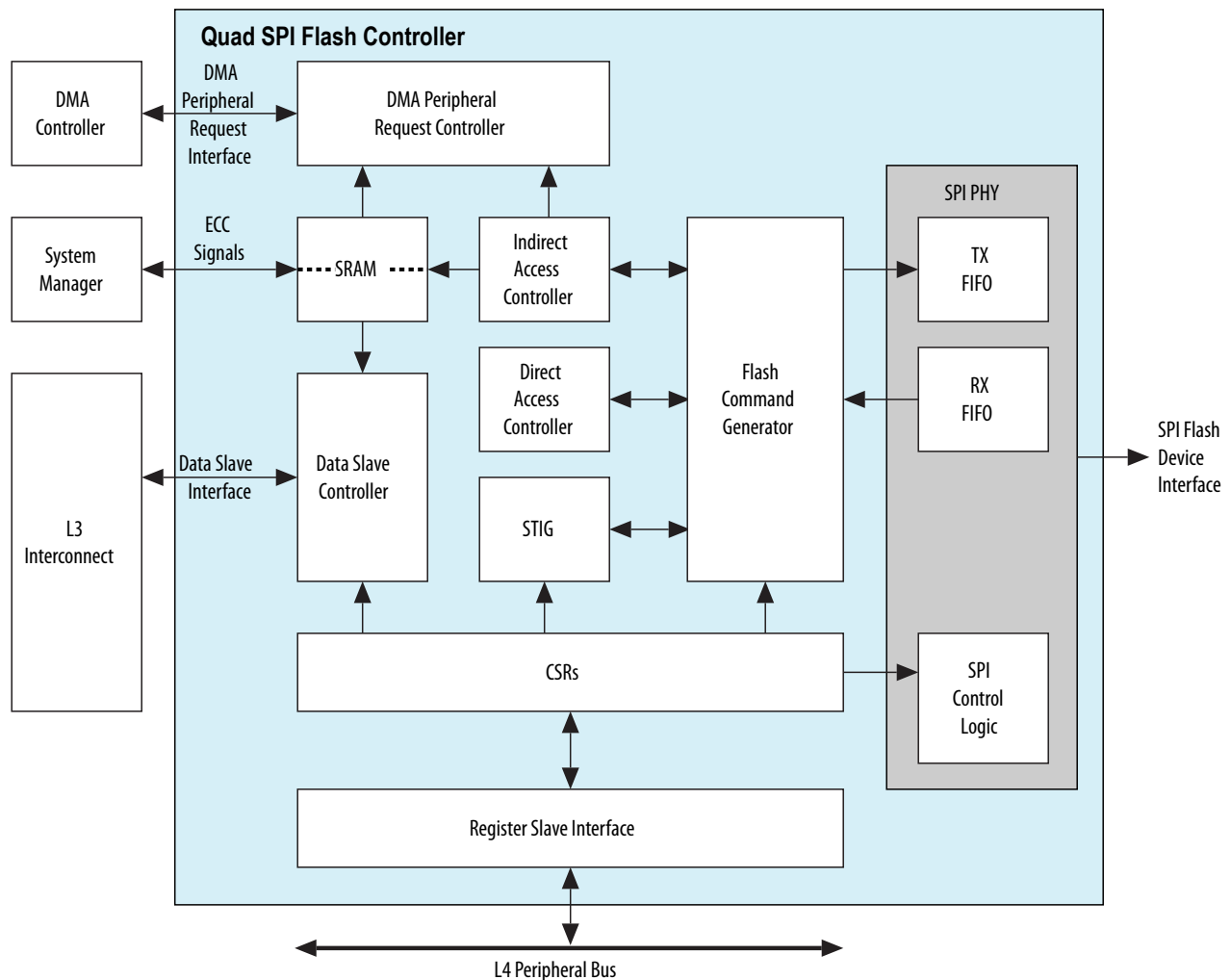
For more information, refer to the supported QSPI flash devices section on this page.

⁰ The quad SPI controller supports any device frequencies, but this speed is limited by the supported flash devices.

⁽⁴⁹⁾ This QSPI controller does not support Double Transfer Rate (DTR) mode.

Quad SPI Flash Controller Block Diagram and System Integration

Figure 15-1: Quad SPI Flash Controller Block Diagram and System Integration



The quad SPI controller consists of the following blocks and interfaces:

- Register slave interface—Slave interface that provides access to the control and status registers (CSRs)
- Data slave controller—Slave interface and controller that provides the following functionality:
 - Performs data transfers to and from the level 3 (L3) interconnect
 - Validates incoming accesses
 - Performs byte or half-word reordering
 - Performs write protection
 - Forwards transfer requests to direct and indirect controller
- Direct access controller—provides memory-mapped slaves direct access to the flash memory
- Indirect access controller—provides higher-performance access to the flash memory through local buffering and software transfer requests

- Software triggered instruction generator (STIG)—generates flash commands through the flash command register (flashcmd) and provides low-level access to flash memory
- Flash command generator—generates flash command and address instructions based on instructions from the direct and indirect access controllers or the STIG
- DMA peripheral request controller—issues requests to the DMA peripheral request interface to communicate with the external DMA controller
- SPI PHY—serially transfers data and commands to the external SPI flash devices

Interface Signals

The quad SPI controller provides four chip select outputs to allow control of up to four external quad SPI flash devices. The outputs serve different purposes depending on whether the device is used in single, dual, or quad operation mode. The following table lists the I/O pin use of the quad SPI controller interface signals for each operation mode.

Table 15-1: Interface Signals

Signal	Mode	Direction	Function
data[0]	Single	Output	Data output 0
	Dual or quad	Bidirectional	Data I/O 0
data[1]	Single	Input	Data input 0
	Dual or quad	Bidirectional	Data I/O 1
data[2]	Single or dual	Output	Active low write protect
	Quad	Bidirectional	Data I/O 2
data[3]	Single, dual, or quad	Bidirectional	Data I/O 3
ss_n[0]	Single, dual, or quad	Output	Active low slave select 0
ss_n[1]			Active low slave select 1
ss_n[2]			Active low slave select 2
ss_n[3]			Active low slave select 3
sclk	Single, dual, or quad	Output	Serial Clock

Functional Description of the Quad SPI Flash Controller

Overview

The quad SPI flash controller uses the register slave interface to select the operation modes and configure the data slave interface for data transfers. The quad SPI flash controller uses the data slave interface for direct and indirect accesses, and the register slave interface for software triggered instruction generator (STIG) operation and SPI legacy mode accesses.

Accesses to the data slave are forwarded to the direct or indirect access controller. If the access address is within the configured indirect address range, the access is sent to the indirect access controller.

Data Slave Interface

The quad SPI flash controller uses the data slave interface for direct, indirect, and SPI legacy mode accesses.

The data slave interface is 32 bits wide and permits byte, half-word, and word accesses. For write accesses, incrementing burst lengths of 1, 4, 8 and 16 are supported. For read accesses, all burst types and sizes are supported.

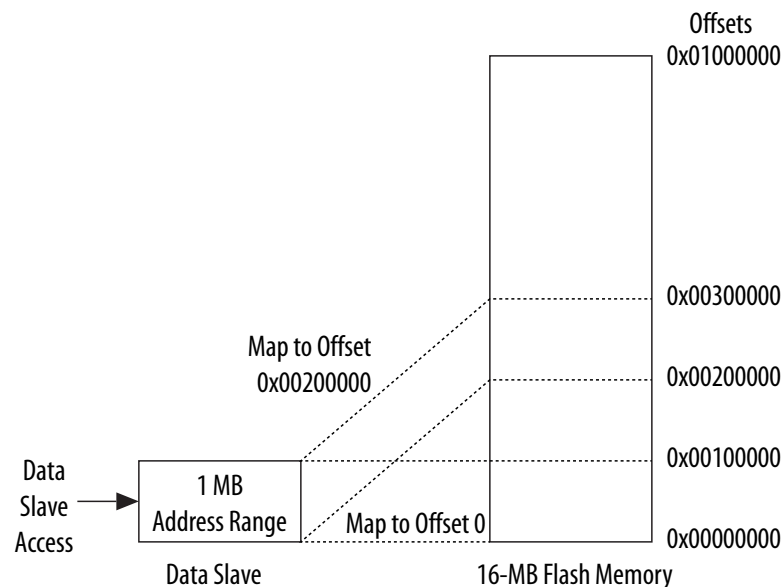
Direct Access Mode

Direct access mode is memory mapped and can be used to both access and directly execute code from external FLASH memory. Any incoming AMBA data slave interface access that is not recognized as being within the programmable indirect trigger region is assumed to be a direct access and is serviced by the direct access controller.

Note: Accesses that use the direct access controller do not use the embedded SRAM.

Data Slave Remapping Example

Figure 15-2: Data Slave Remapping Example



To remap the data slave to access other 1 MB regions of the flash device, enable address remapping in the enable Arm AMBA advanced high speed bus (AHB) address remapping field (enahbremap) of the cfg register. All incoming data slave accesses remap to the offset specified in the remap address register (remapaddr).

The 20 LSBs of incoming addresses are used for accessing the 1 MB region and the higher bits are ignored.

Note: The quad SPI controller does not issue any error status for accesses that lie outside the connected flash memory space.

AHB

The data slave interface is throttled as the read or write burst is carried out. The latency is designed to be as small as possible and is kept to a minimum when the use of XIP read instructions are enabled.

FLASH erase operations, which may be required before a page write, are triggered by software using the documented programming interface. They are not issued automatically.

Once a page program cycle has been started, the QSPI Flash Controller will automatically poll for the write cycle to complete before allowing any further data slave interface accesses to complete. This is achieved by holding any subsequent AHB direct accesses in wait state.

Indirect Access Mode

In indirect access mode, flash data is temporarily buffered in the quad SPI controller's static RAM (SRAM). Software controls and triggers indirect accesses through the register slave interface. The controller transfers data through the data slave interface.

Indirect Read Operation

An indirect read operation reads data from the flash memory, places the data into the SRAM, and transfers the data to an external master through the data slave interface. The indirect read operations are controlled by the following registers:

- Indirect read transfer register (`indr`)
- Indirect read transfer watermark register (`indrwater`)
- Indirect read transfer start address register (`indrstaddr`)
- Indirect read transfer number bytes register (`indrcnt`)
- Indirect address trigger register (`indaddrtrig`)

These registers need to be configured prior to issuing indirect read operations. The start address needs to be defined in the `indrstaddr` register and the total number of bytes to be fetched is specified in the `indrcnt` register. Writing 1 to the start indirect read bit (`start`) of the `indr` register triggers the indirect read operation from the flash memory to populate the SRAM with the returned data.

To read data from the flash device into the SRAM, an external master issues 32-bit read transactions to the data slave interface. The address of the read access must be in the indirect address range. You can configure the indirect address through the `indaddrtrig` register. The external master can issue 32-bit reads until the last word of an indirect transfer. On the final read, the external master may issue a 32-bit, 16-bit or 8-bit read to complete the transfer. If there are less than four bytes of data to read on the last transfer, the external master can still issue a 32-bit read and the quad SPI controller pads the upper bits of the response data with zeros.

Assuming the requested data is present in the SRAM at the time the data slave read is received by the quad SPI controller, the data is fetched from SRAM and the response to the read burst is achieved with minimum latency. If the requested data is not immediately present in the SRAM, the data slave interface enters a wait state until the data has been read from flash memory into SRAM. Once the data has been read from SRAM by the external master, the quad SPI controller frees up the associated resource in the SRAM. If the SRAM is full, reads on the SPI interface are backpressured until space is available in the SRAM. The quad SPI controller completes any current read burst, waits for SRAM to free up, and issues a new read burst at the address where the previous burst was terminated.

The processor can also use the SRAM fill level in the SRAM fill register (`sramfill`) to control when data should be fetched from the SRAM.

Alternatively, you can configure the fill level watermark of the SRAM in the `indrwater` register. When the SRAM fill level passes the watermark level, the indirect transfer watermark interrupt is generated. You can disable this watermark feature by writing a value of all zeroes to the `indrwater` register.

For the final bytes of data read by the quad SPI controller and placed in the SRAM, if the watermark level is greater than zero, the indirect transfer watermark interrupt is generated even when the actual SRAM fill level has not risen above the watermark.

If the address of the read access is outside the range of the indirect trigger address, one of the following actions occurs:

- When direct access mode is enabled, the read uses direct access mode.
- When direct access mode is disabled, the slave returns an error back to the requesting master.

You can cancel an indirect operation by setting the cancel indirect read bit (`cancel`) of the `indr` register to 1. For more information, refer to the “Indirect Read Operation with DMA Disabled” section.

Related Information

[Indirect Read Operation with DMA Disabled](#) on page 15-16

Indirect Write Operation

An indirect write operation programs data from the SRAM to the flash memory. The indirect write operations are controlled by the following registers:

- Indirect write transfer register (`indwr`)
- Indirect write transfer watermark register (`indwrwater`)
- Indirect write transfer start address register (`indwrstaddr`)
- Indirect write transfer number bytes register (`indwrcnt`)
- `indaddrtrig` register

These registers need to be configured prior to issuing indirect write operations. The start address needs to be defined in the `indwrstaddr` register and the total number of bytes to be written is specified in the `indwrcnt` register. The start indirect write bit (`start`) of the `indwr` register triggers the indirect write operation from the SRAM to the flash memory.

To write data from the SRAM to the flash device, an external master issues 32-bit write transactions to the data slave. The address of the write access must be in the indirect address range. You can configure the indirect address through the `indaddrtrig` register. The external master can issue 32-bit writes until the last word of an indirect transfer. On the final write, the external master may issue a 32-bit, 16-bit or 8-bit write to complete the transfer. If there are less than four bytes of data to write on the last transfer, the external master can still issue a 32-bit write and the quad SPI controller discards the extra bytes.

The SRAM size can limit the amount of data that the quad SPI controller can accept from the external master. If the SRAM is not full at the point of the write access, the data is pushed to the SRAM with minimum latency. If the external master attempts to push more data to the SRAM than the SRAM can accept, the quad SPI controller backpressures the external master with wait states. When the SRAM resource is freed up by pushing the data from SRAM to the flash memory, the SRAM is ready to receive more data from the external master. When the SRAM holds an equal or greater number of bytes than the size of a flash page, or when the SRAM holds all the remaining bytes of the current indirect transfer, the quad SPI controller initiates a write operation to the flash memory.

The processor can also use the SRAM fill level, in the `sramfill` register, to control when to write more data into the SRAM.

Alternatively, you can configure the fill level watermark of the SRAM in the `indwrwater` register. When the SRAM fill level falls below the watermark level, an indirect transfer watermark interrupt is generated to tell the software to write the next page of data to the SRAM. Because the quad SPI controller initiates

non-end-of-data writes to the flash memory only when the SRAM contains a full flash page of data, you must set the watermark level to a value greater than one flash page to avoid the system stalling. You can disable this watermark feature by writing a value of all ones to the `indwrwater` register.

If the address of the write access is outside the range of the indirect trigger address, one of the following actions occurs:

- When direct access mode is enabled, the write uses direct access mode.
- When direct access mode is disabled, the slave returns an error back to the requesting master.

You can cancel an indirect operation by setting the cancel indirect write bit (`cancel`) of the `indwr` register to 1. For more information, refer to the “Indirect Write Operation with DMA Disabled” section.

Related Information

[Indirect Write Operation with DMA Disabled](#) on page 15-17

Consecutive Reads and Writes

It is possible to trigger two indirect operations at a time by triggering the `start` bit of the `indrd` or `indwr` register twice in short succession. The second operation can be triggered while the first operation is in progress. For example, software may trigger an indirect read or write operation while an indirect write operation is in progress. The corresponding start and count registers must be configured properly before software triggers each transfer operation.

This approach allows for a short turnaround time between the completion of one indirect operation and the start of a second operation. Any attempt to queue more than two operations causes the indirect read reject interrupt to be generated.

SPI Legacy Mode

SPI legacy mode allows software to access the internal TX FIFO and RX FIFO buffers directly, thus bypassing the direct, indirect and STIG controllers. Software accesses the TX FIFO and RX FIFO buffers by writing any value to any address through the data slave while legacy mode is enabled. You can enable legacy mode with the legacy IP mode enable bit (`enlegacyip`) of the `cfg` register.

Legacy mode allows the user to issue any flash instruction to the flash device, but imposes a heavy software overhead in order to manage the fill levels of the FIFO buffers effectively. The legacy SPI mode is bidirectional in nature, with data continuously being transferred both directions while the chip select is enabled. If the driver only needs to read data from the flash device, dummy data must be written to ensure the chip select stays active, and vice versa for write transactions.

For example, to perform a basic read of four bytes to a flash device that has three address bytes, software must write a total of eight bytes to the TX FIFO buffer. The first byte would be the instruction opcode, the next three bytes are the address, and the final four bytes would be dummy data to ensure the chip select stays active while the read data is returned. Similarly, because eight bytes were written to the TX FIFO buffer, software should expect eight bytes to be returned in the RX FIFO buffer. The first four bytes of this would be discarded, leaving the final four bytes holding the data read from the device.

Because the TX FIFO and RX FIFO buffers are four bytes deep each, software must maintain the FIFO buffer levels to ensure the TX FIFO buffer does not underflow and the RX FIFO buffer does not overflow. Interrupts are provided to indicate when the fill levels pass the watermark levels, which are configurable through the TX threshold register (`txtresh`) and RX threshold register (`rxtresh`).

Register Slave Interface

The quad SPI flash controller uses the register slave interface to configure the quad SPI controller through the quad SPI configuration registers, and to access flash memory under software control, through the `flashcmd` register in the STIG.

STIG Operation

The Software Triggered Instruction Generator (STIG) is used to access the volatile and non-volatile configuration registers, the legacy SPI status register, and other status and protection registers. The STIG also is used to perform ERASE functions. The direct and indirect access controllers are used only to transfer data. The `flashcmd` register uses the following parameters to define the command to be issued to the flash device:

- Instruction opcode
- Number of address bytes
- Number of dummy bytes
- Number of write data bytes
- Write data
- Number of read data bytes

The address is specified through the flash command address register (`flashcmdaddr`). Once these settings have been specified, software can trigger the command with the execute command field (`execcmd`) of the `flashcmd` register and wait for its completion by polling the command execution status bit (`cmdexecstat`) of the `flashcmd` register. A maximum of eight data bytes may be read from the flash command read data lower (`flashcmdrddatalo`) and flash command read data upper (`flashcmdrddataup`) registers or written to the flash command write data lower (`flashcmdwrdatalo`) and flash command write data upper (`flashcmdwrdataup`) registers per command.

Commands issued through the STIG have a higher priority than all other read accesses and therefore interrupt any read commands being requested by the direct or indirect controllers. However, the STIG does not interrupt a write sequence that may have been issued through the direct or indirect access controller. In these cases, it might take a long time for the `cmdexecstat` bit of the `flashcmd` register indicates the operation is complete.

Note: Intel recommends using the STIG instead of the SPI legacy mode to access the flash device registers and perform erase operations.

Local Memory Buffer

The SRAM local memory buffer is a 128 by 32-bit (512 total bytes) memory and includes support for error correction code (ECC). The ECC logic provides outputs to notify the system manager when single-bit correctable errors are detected (and corrected) and when double-bit uncorrectable errors are detected. The ECC logic also allows the injection of single- and double-bit errors for test purposes.

You must initialize memory data before enabling ECC to prevent spurious ECC interrupts when you enable ECC for the first time.

Follow these steps to enable ECC:

1. Turn on the ECC hardware, but disable the interrupts.
2. Initialize the SRAM in the NAND.
3. Clear the ECC event bits, because these bits may have become asserted after Step 1.
4. Enable the ECC interrupts now that the ECC bits have been set.

The SRAM has two partitions, with the lower partition reserved for indirect read operations and the upper partition reserved for indirect write operations, as shown in [Figure 15-1](#). The size of each partition is specified in the SRAM partition register (`srampart`), based on 32-bit word sizes. The number of locations allocated to indirect read is (`srampart + 1`); and the number of locations allocated to indirect write is (`128 - srampart`)⁽⁵⁰⁾. For example, to specify four bytes of storage, write the value 1. The value written to the indirect read partition size field (`addr`) defines the number of entries reserved for indirect read operations. For example, write the value 32 (0x20) to partition the 128-entry SRAM to 32 entries (25%) for read usage and 96 entries (75%) for write usage.

Related Information

[System Manager](#) on page 6-1

DMA Peripheral Request Controller

The DMA peripheral request controller is only used for the indirect mode of operation where data is temporarily stored in the SRAM. The quad SPI flash controller uses the DMA peripheral request interface to trigger the external DMA into performing data transfers between memory and the quad SPI controller.

There are two DMA peripheral request interfaces, one for indirect reads and one for indirect writes. The DMA peripheral request controller can issue two types of DMA requests, single or burst, to the external DMA. The number of bytes for each single or burst request is specified in the number of single bytes (`numsglreqbytes`) and number of burst bytes (`numburstreqbytes`) fields of the DMA peripheral register (`dmaper`). The DMA peripheral request controller splits the total amount of data to be transferred into a number of DMA burst and single requests by dividing the total number of bytes by the number of bytes specified in the burst request, and then dividing the remainder by the number of bytes in a single request.

Note: When programming the DMA controller, the burst request size must match the burst request size set in the quad SPI controller to avoid quickly reaching an overflow or underflow condition.

For indirect reads, the DMA peripheral request controller only issues DMA requests after the data has been retrieved from flash memory and written to SRAM. The rate at which DMA requests are issued depends on the watermark level. The `indrdwater` register defines the minimum fill level in bytes at which the DMA peripheral request controller can issue the DMA request. The higher this number is, the more data that must be buffered in SRAM before the external DMA moves the data. When the SRAM fill level passes the watermark level, the transfer watermark reached interrupt is generated.

For example, consider the following conditions:

- The total amount of data to be read using indirect mode is 256 bytes
- The SRAM watermark level is set at 128 bytes
- Software configures the burst type transfer size to 64 bytes

Under these conditions, the DMA peripheral request controller issues the first DMA burst request when the SRAM fill level passes 128 bytes (the watermark level). The DMA peripheral request controller triggers consecutive DMA burst requests as long as there is sufficient data in the SRAM to perform burst type requests. In this example, DMA peripheral request controller can issue at least two consecutive DMA burst requests to transfer a total of 128 bytes. If there is sufficient data in the SRAM, the DMA peripheral request controller requests the third DMA burst immediately. Otherwise the DMA peripheral request controller waits for the SRAM fill level to pass the watermark level again to trigger the next burst request. When the watermark level is triggered, there is sufficient data in the SRAM to perform the third and fourth burst requests to complete the entire transaction.

For the indirect writes, the DMA peripheral request controller issues DMA requests immediately after the transfer is triggered and continues to do so until the entire indirect write transfer has been transferred. The

⁽⁵⁰⁾ You cannot allocate all of the SRAM for writes.

rate at which DMA requests are issued depends on the watermark level. The `indwrwater` register defines the maximum fill level in bytes at which the controller can issue the first DMA burst or single request. When the SRAM fill level falls below the watermark level, the transfer watermark reached interrupt is generated. When there is one flash page of data in the SRAM, the quad SPI controller initiates the write operation from SRAM to the flash memory.

Software can disable the DMA peripheral request interface with the `endma` field of the `cfg` register. If a master other than the DMA performs the data transfer for indirect operations, the DMA peripheral request interface must be disabled. By default, the indirect watermark registers are set to zero, which means the DMA peripheral request controller can issue DMA request as soon as possible.

For more information about the HPS DMA controller, refer to the *DMA Controller* chapter in volume 3 of the Arria® V Device Handbook.

Related Information

[DMA Controller](#) on page 16-1

Arbitration between Direct/Indirect Access Controller and STIG

When multiple controllers are active simultaneously, a fixed-priority arbitration scheme is used to arbitrate between each interface and access the external FLASH. The fixed priority is defined as follows, highest priority first.

1. The Indirect Access Write
2. The Direct Access Write
3. The STIG
4. The Direct Access Read
5. The Indirect Access Read

Each controller is back pressured while waiting to be serviced.

Configuring the Flash Device

For read and write accesses, software must initialize the device read instruction register (`devrd`) and the device write instruction register (`devwr`). These registers include fields to initialize the instruction opcodes that should be used as well as the instruction type, and whether the instruction uses single, dual or quad pins for address and data transfer. To ensure the quad SPI controller can operate from a reset state, the opcode registers reset to opcodes compatible with single I/O flash devices.

The quad SPI flash controller uses the instruction transfer width field (`instwidth`) of the `devrd` register to set the instruction transfer width for both reads and writes. There is no `instwidth` field in the `devwr` register. If instruction type is set to dual or quad mode, the address transfer width (`addrwidth`) and data transfer width (`datawidth`) fields of both registers are redundant because the address and data type is based on the instruction type. Thus, software can support the less common flash instructions where the opcode, address, and data are sent on two or four lanes. For most instructions, the opcodes are sent serially to the flash device, even for dual and quad instructions. One of the flash devices that supports instructions that can send the opcode over two or four lanes is the Micron N25Q128. For reference, [Table 15-2](#) and [Table 15-3](#) show how software should configure the quad SPI controller for each specific read and write instruction, respectively, supported by the Micron N25Q128 device.

Table 15-2: Quad SPI Configuration for Micron N25Q128 Device (Read Instructions)

Instruction	Lanes Used By Opcode	Lanes Used to Send Address	Lanes Used to Send Data	instwidth Value	addrwidth Value	datawidth Value
Read	1	1	1	0	0	0
Fast read	1	1	1	0	0	0
Dual output fast read (DOFR)	1	1	2	0	0	1
Dual I/O fast read (DIOFR)	1	2	2	0	1	1
Quad output fast read (QOFR)	1	1	4	0	0	2
Quad I/O fast read (QIOFR)	1	4	4	0	2	2
Dual command fast read (DCFR)	2	2	2	1	Don't care	Don't care
Quad command fast read (QCFR)	4	4	4	2	Don't care	Don't care

Table 15-3: Quad SPI Configuration for Micron N25Q128 Device (Write Instructions)

Instruction	Lanes Used By Opcode	Lanes Used to Send Address	Lanes Used to Send Data	instwidth Value	addrwidth Value	datawidth Value
Page program	1	1	1	0	0	0
Dual input fast program (DIFP)	1	1	2	0	0	1
Dual input extended fast program (DIEFP)	1	2	2	0	1	1
Quad input fast program (QIFP)	1	1	4	0	0	2

Instruction	Lanes Used By Opcode	Lanes Used to Send Address	Lanes Used to Send Data	instwidth Value	addrwidth Value	datawidth Value
Quad input extended fast program (QIEFP)	1	4	4	0	2	2
Dual command fast program (DCFP)	2	2	2	1	Don't care	Don't care
Quad command fast program (QCFP)	4	4	4	2	Don't care	Don't care

Write Request

The Write Enable Latch (WEL) bit within the flash device itself must be high before a write sequence can be issued. The command generator automatically issues the Write Enable (WREN) instruction to set the WEL bit before triggering a write command through the direct or indirect access controllers. When write requests are no longer received and all outstanding requests have been sent, then the flash device starts the page program write cycle. Any incoming request at this time is held in wait states until the cycle has completed. The controller automatically polls the flash device Read Status Register (RDSR) to identify when the write cycle has completed. This continues until the Write In Progress bit and WEL bit have cleared to zero. The op-code for the WREN instruction is typically 0x06h and 0x05h for the RDSR instruction.

Note: These op-codes are common between devices.

XIP Mode

The quad SPI controller supports XIP mode, if the flash devices support XIP mode. Depending on the flash device, XIP mode puts the flash device in read-only mode, reducing command overhead.

The quad SPI controller must instruct the flash device to enter XIP mode by sending the mode bits. When the enter XIP mode on next read bit (enterxipnextrd) of the cfg register is set to 1, the quad SPI controller and the flash device are ready to enter XIP mode on the next read instruction. When the enter XIP mode immediately bit (enterxipimm) of the cfg register is set to 1, the quad SPI controller and flash device enter XIP mode immediately.

When the enterxipnextrd or enterxipimm bit of the cfg register is set to 0, the quad SPI controller and flash device exit XIP mode on the next read instruction. For more information, refer to the “XIP Mode Operations” section.

Related Information

[XIP Mode Operations](#) on page 15-18

Write Protection

You can program the controller to write protect a specific region of the flash device. The protected region is defined as a set of blocks, specified by a starting and ending block. Writing to an area of protected flash region memory generates an error and triggers an interrupt.

You define the block size by specifying the number of bytes per block through the number of bytes per block field (bytespersubsector) of the device size register (devsz). The lower write protection register (lowwrprot) specifies the first flash block in the protected region. The upper write protection register (upwrprot) specifies the last flash block in the protected region.

The write protection enable bit (en) of the write protection register (wrprot) enables and disables write protection. The write protection inversion bit (inv) of the wrprot register flips the definition of protection so that the region specified by lowwrprt and upwrprt is unprotected and all flash memory outside that region is protected.

Data Slave Sequential Access Detection

The quad SPI flash controller detects sequential accesses to the data slave interface by comparing the current access with the previous access. An access is sequential when it meets the following conditions:

- The address of the current access sequentially follows the address of the previous access.
- The direction of the current access (read or write) is the same as previous access.
- The size of the current access (byte, half-word, or word) is the same as previous access.

When the access is detected as nonsequential, the sequential access to the flash device is terminated and a new sequential access begins. Intel recommends accessing the data slave sequentially. Sequential access has less command overhead, and therefore, increases data throughput.

Clocks

There are two clock inputs to the quad SPI controller: `l4_mp_clk` and `qspi_ref_clk`; and one clock output: `qspi_clk`. The quad SPI flash controller uses the `l4_mp_clk` clock to clock the data slave transfers and register slave accesses. The `qspi_ref_clk` clock is the reference clock for the quad SPI controller and is used to serialize the data and drive the external SPI interface. The `qspi_clk` clock is the clock source for the connected flash devices.

The following is true for the following reference clocks:

- `qspi_ref_clk` should be greater than `l4_mp_clk`
- `qspi_ref_clk` must be greater than two times `qspi_clk`

The `qspi_clk` clock is derived by dividing down the `qspi_ref_clk` clock by the baud rate divisor field (bauddiv) of the `cfg` register.

Related Information

[Clock Manager](#) on page 3-1

Resets

A single reset signal (`qspi_flash_rst_n`) is provided as an input to the quad SPI controller. The reset manager drives the signal on a cold or warm reset.

Related Information

[Reset Manager](#) on page 4-1

Taking the Quad SPI Flash Controller Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

SRAM Initialization Procedure with ECC Enabled

1. Set 0x40 (default) to `srampart` register (SRAM_PARTITION_REG).
2. Enable ECC.
3. Fill up the read partition in indirect read transfer mode. Confirm `sramfill.Indrdpart` register is 0x41.
4. Fill up and initialize the write partition in indirect operation. Confirm `sramfill.Indwrpart` register is 0x40.
5. Clear ECC interrupt status register.
6. Enable ECC interrupt.

Note: SRAM Partition Configuration Register defines the size of the indirect read partition in the SRAM. By default, half of the SRAM is reserved for indirect read operation, and half for indirect write.

The number of locations allocated to indirect read = SRAM_PARTITION_REG + 1.

The number of locations allocated to indirect write = 128 - SRAM_PARTITION_REG.

Interrupts

All interrupt sources are combined to create a single level-sensitive, active-high interrupt (`qspi_intr`). Software can determine the source of the interrupt by reading the interrupt status register (`irqstat`). By default, the interrupt source is cleared when software writes a one (1) to the interrupt status register. The interrupts are individually maskable through the interrupt mask register (`irqmask`). [Table 15-4](#) lists the interrupt sources in the `irqstat` register.

Table 15-4: Interrupt Sources in the `irqstat` Register

Interrupt Source	Description
Underflow detected	When 0, no underflow has been detected. When 1, the data slave write data is being supplied too slowly. This situation can occur when data slave write data is being supplied too slowly to keep up with the requested write operation. This bit is reset only by a system reset and cleared only when a 1 is written to it.
Indirect operation complete	The controller has completed a triggered indirect operation.
Indirect read reject	An indirect operation was requested but could not be accepted because two indirect operations are already in the queue.

Interrupt Source	Description
Protected area write attempt	A write to a protected area was attempted and rejected.
Illegal data slave access detected	An illegal data slave access has been detected. Data slave wrapping bursts and the use of split and retry accesses can cause this interrupt. It is usually an indicator that soft masters in the FPGA fabric are attempting to access the HPS in an unsupported way.
Transfer watermark reached	The indirect transfer watermark level has been reached.
Receive overflow	This condition occurs only in legacy SPI mode. When 0, no overflow has been detected. When 1, an over flow to the RX FIFO buffer has occurred. This bit is reset only by a system reset and cleared to zero only when this register is written to. If a new write to the RX FIFO buffer occurs at the same time as a register is read, this flag remains set to 1.
TX FIFO not full	This condition occurs only in legacy SPI mode. When 0, the TX FIFO buffer is full. When 1, the TX FIFO buffer is not full.
TX FIFO full	This condition occurs only in legacy SPI mode. When 0, the TX FIFO buffer is not full. When 1, the TX FIFO buffer is full.
RX FIFO not empty	This condition occurs only in legacy SPI mode. When 0, the RX FIFO buffer is empty. When 1, the RX FIFO buffer is not empty.
RX FIFO full	This condition occurs only in legacy SPI mode. When 0, the RX FIFO buffer is not full. When 1, the RX FIFO buffer is full.
Indirect read partition overflow	Indirect Read Partition of SRAM is full and unable to immediately complete indirect operation

Quad SPI Flash Controller Programming Model

Setting Up the Quad SPI Flash Controller

The following steps describe how to set up the quad SPI controller:

1. Wait until any pending operation has completed.
2. Disable the quad SPI controller with the quad SPI enable field (en) of the cfg register.
3. Update the instwidth field of the devrd register with the instruction type you wish to use for indirect and direct writes and reads.
4. If mode bit enable bit (enmodebits) of the devrd register is enabled, update the mode bit register (modebit).
5. Update the devsz register as needed.
Parts or all of this register might have been updated after initialization. The number of address bytes is a key configuration setting required for performing reads and writes. The number of bytes per page is required for performing any write. The number of bytes per device block is only required if the write protect feature is used.
6. Update the device delay register (delay).
This register allows the user to adjust how the chip select is driven after each flash access. Each device may have different timing requirements. If the serial clock frequency is increased, these timing requirements become more critical. The numbers specified in this register are based on the period of the qspi_ref_clk clock. For example, some devices need 50 ns minimum time before the slave select can be reasserted after it has been deasserted. When the device is operating at 100 MHz, the clock period is 10 ns, so 40 ns extra is required. If the qspi_ref_clk clock is running at 400 MHz (2.5 ns period), specify a value of at least 16 to the clock delay for chip select deassert field (nss) of the delay register.
7. Update the remapaddr register as needed.
This register only affects direct access mode.
8. Set up and enable the write protection registers (wrprot, lowwrprot, and upwrprot) when write protection is required.
9. Enable required interrupts through the irqmask register.
10. Set up the bauddiv field of the cfg register to define the required clock frequency of the target device.
11. Update the read data capture register (rddatacap) if you need to change the auto-filled value.
This register delays when the read data is captured and can help when the read data path from the device to the quad SPI controller is long and the device clock frequency is high.
12. Enable the quad SPI controller with the en field of the cfg register.

Related Information

[Intel Arria 10 Device Datasheet](#)

Indirect Read Operation with DMA Disabled

The following steps describe the general software flow to set up the quad SPI controller for indirect read operation with the DMA disabled:

1. Perform the steps described in the [Setting Up the Quad SPI Flash Controller](#) on page 15-16 section.
2. Set the flash memory start address in the indrdstaddr register.
3. Set the number of bytes to be transferred in the indrdcnt register.
4. Set the indirect transfer trigger address in the indaddrtrig register.
5. Set up the required interrupts through the irqmask register.
6. If the watermark level is used, set the SRAM watermark level through the indrdwater register.

7. Start the indirect read operation by setting the `start` field of the `indr` register to 1.
8. Either use the watermark level interrupt or poll the SRAM fill level in the `sramfill` register to determine when there is sufficient data in the SRAM.
9. Issue a read transaction to the indirect address to access the SRAM. Repeat [step 8](#) if more read transactions are needed to complete the indirect read transfer.
10. Either use the indirect complete interrupt to determine when the indirect read operation has completed or poll the completion status of the indirect read operation through the indirect completion status bit (`ind_ops_done_status`) of the `indr` register.

Related Information

[Setting Up the Quad SPI Flash Controller](#) on page 15-16

Indirect Read Operation with DMA Enabled

The following steps describe the general software flow to set up the quad SPI controller for indirect read operation with the DMA enabled:

1. Perform the steps described in the [Setting Up the Quad SPI Flash Controller](#) on page 15-16 section.
2. Set the flash memory start address in the `indrstaddr` register.
3. Set the number of bytes to be transferred in the `indrcnt` register.
4. Set the indirect transfer trigger address in the `indaddrtrig` register.
5. Set the number of bytes for single and burst type DMA transfers in the `dmaper` register.
6. Optionally set the SRAM watermark level in the `indrwater` register to control the rate DMA requests are issued.
7. Start an indirect read access by setting the `start` field of the `indr` register to 1.
8. Either use the indirect complete interrupt to determine when the indirect read operation has completed or poll the completion status of the indirect read operation through the `ind_ops_done_status` field of the `indr` register.

Related Information

[Setting Up the Quad SPI Flash Controller](#) on page 15-16

Indirect Write Operation with DMA Disabled

The following steps describe the general software flow to set up the quad SPI controller for indirect write operation with the DMA disabled:

1. Perform the steps described in the [Setting Up the Quad SPI Flash Controller](#) on page 15-16 section.
2. Set the flash memory start address in the `indwrstaddr` register.
3. Set up the number of bytes to be transferred in the `indwrcnt` register.
4. Set the indirect transfer trigger address in the `indaddrtrig` register.
5. Set up the required interrupts through the interrupt mask register (`irqmask`).
6. Start the indirect write operation by setting the `start` field of the `indwr` register to 1.
7. Either use the watermark level interrupt or poll the SRAM fill level in the `sramfill` register to determine when there is sufficient space in the SRAM.
8. Issue a write transaction to the indirect address to write one flash page of data to the SRAM. Repeat [step 8](#) if more write transactions are needed to complete the indirect write transfer. The final write may be less than one page of data.

Related Information

- [Indirect Write Operation](#) on page 15-6
- [Setting Up the Quad SPI Flash Controller](#) on page 15-16

Indirect Write Operation with DMA Enabled

The following steps describe the general software flow to set up the quad SPI controller for indirect write operation with the DMA enabled:

1. Perform the steps described in the [Setting Up the Quad SPI Flash Controller](#) on page 15-16 section.
2. Set the flash memory start address in the `indwrstaddr` register.
3. Set the number of bytes to be transferred in the `indcnt` field of the `indwr` register.
4. Set the indirect transfer trigger address in the `indaddrtrig` register.
5. Set the number of bytes for single and burst type DMA transfers in the `dmaper` register.
6. Optionally set the SRAM watermark level in the `indwrwater` register to control the rate DMA requests are issued. The value set must be greater than one flash page. For more information, refer to the [Indirect Write Operation](#) on page 15-6 section.
7. Start the indirect write access by setting the `start` field of the `indirwr` register to 1.
8. Either use the indirect complete interrupt to determine when the indirect write operation has completed or poll the completion status of the indirect write operation through the `ind_ops_done_status` field of the `indwr` register.

Related Information

- [Indirect Write Operation](#) on page 15-6
- [Setting Up the Quad SPI Flash Controller](#) on page 15-16

XIP Mode Operations

XIP mode is supported in most SPI flash devices. However, flash device manufacturers do not use a consistent standard approach. Most use signature bits that are sent to the device immediately following the address bytes. Some devices use signature bits and also require a flash device configuration register write to enable XIP mode.

Entering XIP Mode

Micron Quad SPI Flash Devices with Support for Basic-XIP

To enter XIP mode in a Micron quad SPI flash device with support for Basic-XIP, perform the following steps:

1. Save the values in the mode bits, if you intend to restore them upon exit.
2. Disable the direct access controller and indirect access controller to ensure no new read or write accesses are sent to the flash device.
3. Set the XIP mode bits in the `modebit` register to 0x80.
4. Enable the quad SPI controller's XIP mode by setting the `enterxipnextrd` bit of the `cfg` register to 1.
5. Re-enable the direct access controller and, if required, the indirect access controller.

Micron Quad SPI Flash Devices without Support for Basic-XIP

To enter XIP mode in a Micron quad SPI flash device without support for Basic-XIP, perform the following steps:

1. Save the values in the mode bits, if you intend to restore them upon exit.
2. Disable the direct access controller and indirect access controller to ensure no new read or write accesses gets sent to the flash device.
3. Ensure XIP mode is enabled in the flash device by setting the volatile configuration register (VCR) bit 3 to 1. Use the `flashcmd` register to issue the VCR write command.
4. Set the XIP mode bits in the `modebit` register to 0x00.
5. Enable the quad SPI controller's XIP mode by setting the `enterxipnextrd` bit of the `cfg` register to 1.
6. Re-enable the direct access controller and, if required, the indirect access controller.

Winbond Quad SPI Flash Devices

To enter XIP mode in a Winbond quad SPI flash device, perform the following steps:

1. Save the values in the mode bits, if you intend to restore them upon exit.
2. Disable the direct access controller and indirect access controller to ensure no new read or write accesses are sent to the flash device.
3. Set the XIP mode bits in the `modebit` register to 0x20.
4. Enable the quad SPI controller's XIP mode by setting the `enterxipnextrd` bit of the `cfg` register to 1.
5. Re-enable the direct access controller and, if required, the indirect access controller.

Spansion Quad SPI Flash Devices

To enter XIP mode a Spansion quad SPI flash device, perform the following steps:

1. Save the values in the mode bits, if you intend to restore them upon exit.
2. Disable the direct access controller and indirect access controller to ensure no new read or write accesses are sent to the flash device.
3. Set the XIP mode bits in the `modebit` register to 0xA0.
4. Enable the quad SPI controller's XIP mode by setting the `enterxipnextrd` bit of the `cfg` register to 1.
5. Re-enable the direct access controller and, if required, the indirect access controller.

Exiting XIP Mode

To exit XIP mode, perform the following steps:

1. Disable the direct access controller and indirect access controller to ensure no new read or write accesses are sent to the flash device.
2. Restore the mode bits to the values before entering XIP mode, depending on the flash device and manufacturer.
3. Set the `enterxipnextrd` bit of the `cfg` register to 0.

The flash device must receive a read instruction before it can disable its internal XIP mode state. Thus, XIP mode internally stays active until the next read instruction is serviced. Ensure that XIP mode is disabled before the end of any read sequence.

XIP Mode at Power on Reset

Some flash devices can be XIP-enabled as a nonvolatile configuration setting, allowing the flash device to enter XIP mode at power-on reset (POR) without software intervention. Software cannot discover the XIP state at POR through flash status register reads because an XIP-enabled flash device can only be accessed through the XIP read operation. If you know the device may enter XIP mode at POR, have your initial boot software configure the `modebit` register and set the `enterxipimm` bit of the `cfg` register to 1.

If you do not know in advance whether or not the device may enter XIP mode at POR, have your initial boot software issue an XIP mode exit command through the `flashcmd` register, then follow the steps in the “*Entering XIP Mode*” section. Software must be aware of the mode bit requirements of the device, because XIP mode entry and exit varies by device.

Related Information

[Entering XIP Mode](#) on page 15-18

Quad SPI Flash Controller Address Map and Register Definitions

The address map and register definitions for the Quad SPI Flash Controller consist of the following regions:

- QSPI Flash Controller Module Registers
- QSPI Flash Module Data

Related Information

[Introduction to the Hard Processor System](#) on page 2-1

2021.05.07

av_5v4



Subscribe



Send Feedback

This chapter describes the direct memory access controller (DMAC) contained in the hard processor system (HPS). The DMAC transfers data between memory and peripherals and other memory locations in the system. The DMA controller is an instance of the Arm CoreLink DMA Controller (DMA-330).

Related Information

Arm Information Center

For more information about Arm's DMA-330 controller, refer to the *CoreLink DMA Controller DMA-330 Revision: r1p2* Technical Reference Manual on the Arm Infocenter website.

Features of the DMA Controller

The HPS provides one DMAC to handle the data transfer between memory-mapped peripherals and memories, off-loading this work from the microprocessor unit (MPU) subsystem.

The DMAC supports multiple transfer types:

- Memory-to-memory
- Memory-to-peripheral
- Peripheral-to-memory

The DMAC supports up to:

- Eight logical channels for different levels of service requirements
- 31 peripheral handshake interfaces for peripheral hardware flow control

The DMAC provides:

- An instruction processing block that enables it to process program code that controls a DMA transfer
- An Arm Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) master interface unit to fetch the program code from system memory into its instruction cache

Note: The AXI master interface also performs DMA data transfer. The DMA instruction execution engine executes the program code from its instruction cache and schedules read or write AXI instructions through the respective instruction queues.

- A multi-FIFO (MFIFO) data buffer that stores data that it reads, or writes, during a DMA transfer
- 11 interrupt outputs to enable efficient communication of events to the MPU subsystem

Note: The peripheral request interfaces support the connection of DMA-capable peripherals to enable memory-to-peripheral and peripheral-to-memory transfers to occur, without intervention from the processor.

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

The DMAC supports the following interface protocols:

- Synopsys protocol
 - Serial peripheral interface (SPI)
 - Universal asynchronous receiver/transmitter (UART)
 - Inter-integrated circuit (I²C)
 - FPGA
- Arm protocol
 - Quad SPI flash controller
 - System trace macrocell (STM)

Dual slave interfaces enable the operation of the DMA controller to be partitioned into the secure state and non-secure state. The network interconnect must be configured to ensure that only secure transactions can access the secure interface. The slave interfaces can access status registers and also directly execute instructions in the DMA controller.

The DMAC has the following features:

- A small instruction set that provides a flexible method of specifying the DMA operations. This architecture provides greater flexibility than the fixed capabilities of a Linked-List Item (LLI) based DMA controller.
- Supports multiple transfer types:
 - Memory-to-memory
 - Memory-to-peripheral
 - Peripheral-to-memory
 - Scatter-gather
- Supports up to eight DMA channels.
- Supports up to eight outstanding AXI read and eight outstanding AXI write transactions.
- Enables software to schedule up to 16 outstanding read and 16 outstanding write instructions.
- Supports 11 interrupt lines into the MPU subsystem⁽⁵¹⁾:
 - One for DMA thread abort
 - Eight for events
 - Two for MFIFO buffer ECC
- Single and double bit ECC support
- Supports 31 peripheral request interfaces:
 - Eight for FPGA
 - Four for I²C
 - Four for I²C (EMAC)
 - Eight for SPI
 - Two for quad SPI
 - One for System Trace Macrocell
 - Four for UART

Related Information

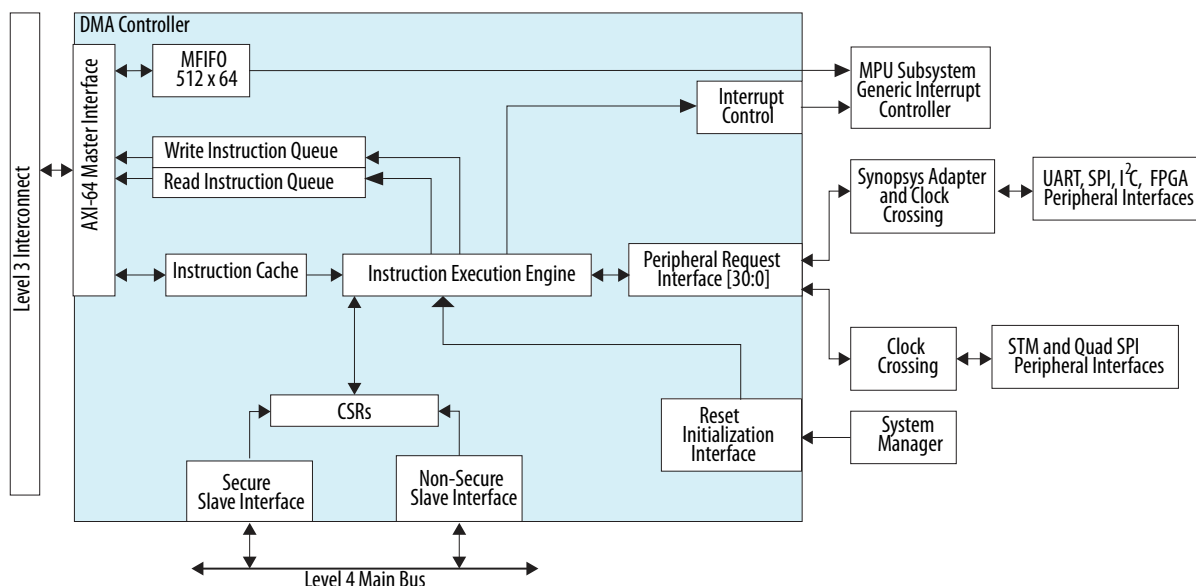
Using Events and Interrupts

⁽⁵¹⁾ For a description of these interrupts, refer to the "Using Events and Interrupts" chapter.

DMA Controller Block Diagram and System Integration

The following figure shows a block diagram of the DMAC and how it integrates into the rest of the HPS system.

Figure 16-1: DMA Controller Connectivity



The `l4_main_clk` clock drives the DMA controller, controller logic, and all the interfaces. The DMA controller accesses the level 3 (L3) main switch with its 64-bit AXI master interface.

The DMA controller provides the following slave interfaces:

- Non-secure slave interface
- Secure slave interface

You can use these slave interfaces to access the registers that control the functionality of the DMA controller. Since the DMA controller supports some peripherals that do not comply with the Arm DMA peripheral interface protocol, some adapters are developed to allow these peripherals to work with the DMAC.

Functional Description of the DMA Controller

This section describes the major interfaces and components of the DMAC and its operation.

The DMAC contains an instruction processing block that processes program code to control a DMA transfer. The program code is stored in a region of system memory that the DMAC accesses using its AXI master interface. The DMAC stores instructions temporarily in an internal cache.

The DMAC has eight DMA channels. Each channel supports a single concurrent thread of DMA operation. In addition, a single DMA manager thread exists, and you can use it to initialize the DMA channel threads.

The DMAC executes one instruction per clock cycle. To ensure that it regularly executes each active thread, the DMAC alternates by processing the DMA manager thread and then a DMA channel thread. It performs a round-robin process when selecting the next active DMA channel thread to execute.

The DMAC uses variable-length instructions that consist of one to six bytes. It provides a separate program counter (PC) register for each DMA channel.

The DMAC includes a 16-line instruction cache to improve the instruction fetch performance. Each instruction cache line contains eight, four-byte words for a total cache line size of 32 bytes. The DMAC instruction cache size is therefore 16 lines times 32 bytes per line which equals 512 bytes. When a thread requests an instruction from an address, the cache performs a lookup. If a cache hit occurs, then the cache immediately provides the instruction. Otherwise, the thread is stalled while the DMAC performs a cache line fill through the AXI master interface. If an instruction spans the end of a cache line, the DMAC performs multiple cache accesses to fetch the instruction.

Note: When a cache line fill is in progress, the DMAC enables other threads to access the cache. But if another cache miss occurs, the pipeline stalls until the first line fill is complete.

When a DMA channel thread executes a load or store instruction, the DMAC adds the instruction to the relevant read or write queue. The DMAC uses these queues as an instruction storage buffer prior to it issuing the instructions on the AXI. The DMAC also contains an MFIFO data buffer in which it stores data that it reads or writes during a DMA transfer.

The DMAC provides multiple interrupt outputs to enable efficient communication of events to the system CPUs. The peripheral request interfaces support the connection of DMA-capable peripherals to enable memory-to-peripheral and peripheral-to-memory DMA transfers to occur without intervention from the microprocessor.

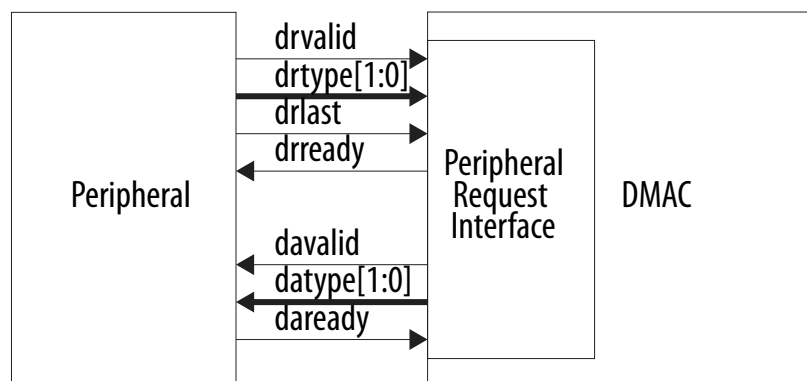
Dual slave interfaces enable the operation of the DMAC to be partitioned into the secure state and non-secure states. You can access status registers and also directly execute instructions in the DMAC with the slave interfaces.

Peripheral Request Interface

The following figure shows that the peripheral request interface consists of a peripheral request bus and a DMAC acknowledge bus that use the prefixes:

- dr—The peripheral request bus
- da—The DMAC acknowledge bus

Figure 16-2: Request and Acknowledge Buses on the Peripheral Request Interface



Both buses use the valid and ready handshake that the AXI protocol describes.

The peripheral uses `drtype[1:0]` to either:

- Request a single transfer
- Request a burst transfer
- Acknowledge a flush request

The peripheral uses `dr last` to notify the DMAC that the request on `drtype[1:0]` is the last request of the DMA transfer sequence. `dr last` is transferred at the same time as `drtype[1:0]`.

The DMAC can indicate the following using `datatype[1:0]`:

- When it completes the requested single transfer
- When it completes the requested burst transfer
- When it issues a flush request

Note: If you configure the DMAC to provide more than one peripheral request interface, each interface is assigned a unique identifier, `_<x>` where `<x>` represents the number of the interface.

For the Synopsys protocol, the following signals are used in the handshaking protocol:

- `dma_tx_req_n`
- `dma_rx_req_n`
- `dma_tx_ack_n`
- `dma_rx_ack_n`
- `dma_tx_single_n`
- `dma_rx_single_n`

Peripheral Request Interface Mapping

You can assign a peripheral request interface to any of the DMA channels. When a DMA channel thread executes DMAWFP, the value programmed in the peripheral [4:0] field specifies the peripheral associated with that DMA channel.

The DMAC supports 32 peripheral request handshakes. Each request handshake can receive up to four outstanding requests, and is assigned a specific peripheral device ID. The following table lists the peripheral device ID assignments.

Table 16-1: Peripheral Request Interface Mapping

Peripheral	Request Interface ID	Protocol
FPGA 0	0	Synopsys
FPGA 1	1	Synopsys
FPGA 2	2	Synopsys
FPGA 3	3	Synopsys
FPGA 4	4	Synopsys
FPGA 5	5	Synopsys



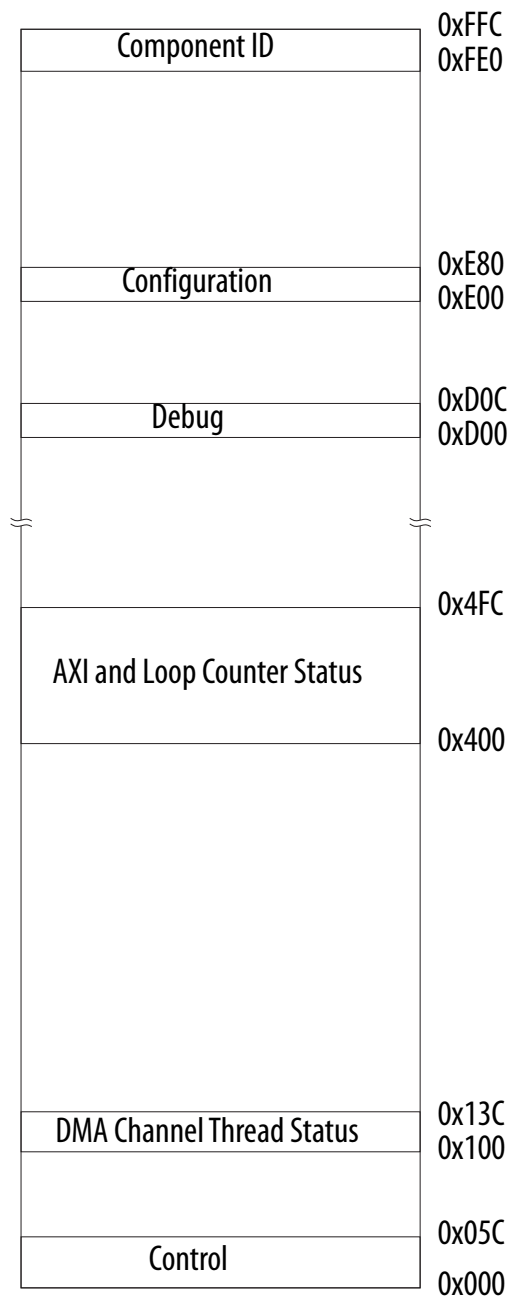
Peripheral	Request Interface ID	Protocol
FPGA 6	6	Synopsys
FPGA 7	7	Synopsys
I ² C 0 Tx	8	Synopsys
I ² C 0 Rx	9	Synopsys
I ² C 1 Tx	10	Synopsys
I ² C 1 Rx	11	Synopsys
I ² C 2 Tx (EMAC)	12	Synopsys
I ² C 2 Rx (EMAC)	13	Synopsys
I ² C 3 Tx (EMAC)	14	Synopsys
I ² C 3 Rx (EMAC)	15	Synopsys
SPI Master 0 Tx	16	Synopsys
SPI Master 0 Rx	17	Synopsys
SPI Slave 0 Tx	18	Synopsys
SPI Slave 0 Rx	19	Synopsys
SPI Master 1 Tx	20	Synopsys
SPI Master 1 Rx	21	Synopsys
SPI Slave 1 Tx	22	Synopsys
SPI Slave 1 Rx	23	Synopsys
Quad SPI Flash Tx	24	Arm
Quad SPI Flash Rx	25	Arm
STM	26	Arm
Reserved	27	N/A
UART 0 Tx	28	Synopsys

Peripheral	Request Interface ID	Protocol
UART 0 Rx	29	Synopsys
UART 1 Tx	30	Synopsys
UART 1 Rx	31	Synopsys

DMA Controller Address Map and Register Definitions

The following DMAC register map spans a 4 KB region, consists of the following sections.

Figure 16-3: DMAC Summary Register Map



- **Control registers**— allow you to control the DMAC.
- **DMA channel thread status registers**—provide the status of the DMA channel threads.
- **AXI and loop counter status registers**—provide the AXI transfer status and the loop counter status for each DMA channel thread.
- **Debug registers**—enable the following functionality:
 - Allow you to send instructions to a thread when debugging the program code.
 - Allow system firmware to send instructions to the DMA manager thread.
- **Configuration registers**—enable system firmware to identify the configuration of the DMAC and control the behavior of the watchdog.
- **Component ID registers**— enable system firmware to identify peripherals. Do not attempt to access reserved or unused address locations. Attempting to access these locations can result in an unpredictable behavior.

Address Map and Register Definitions

The address map and register definitions for the DMA Controller consist of the following regions:

- Nonsecure DMA Module Address Space
- Secure DMA Module Address Space

Related Information

[Introduction to the Hard Processor System](#) on page 2-1

The base addresses of all modules are also listed in the *Introduction to the Hard Processor* chapter.

Ethernet Media Access Controller 17

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) provides two Ethernet media access controller (EMAC) peripherals. Each EMAC can be used to transmit and receive data at 10/100/1000 Mbps over Ethernet connections in compliance with the IEEE 802.3 specification. The EMACs are instances of the Synopsys DesignWare 3504-0 Universal 10/100/1000 Ethernet MAC (DWC_gmac).

The EMAC has an extensive memory-mapped control and status register (CSR) set, which can be accessed by the Arm Cortex-A9 MPCore.

For an understanding of this chapter, you should be familiar with the basics of IEEE 802.3 media access control (MAC). ⁽⁵²⁾

Related Information

IEEE Standards Association

For complete information about IEEE 802.3 MAC, refer to the *IEEE 802.3 2008 Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, available on the IEEE Standards Association website.

⁽⁵²⁾ Portions © 2017 Synopsys, Inc. Used with permission. All rights reserved. Synopsys & DesignWare are registered trademarks of Synopsys, Inc. All documentation is provided "as is" and without any warranty. Synopsys expressly disclaims any and all warranties, express, implied, or otherwise, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, and any warranties arising out of a course of dealing or usage of trade.

†Paragraphs marked with the dagger (†) symbol are Synopsys Proprietary. Used with permission.

Features of the Ethernet MAC

MAC

- IEEE 802.3-2008 compliant
- Data rates of 10/100/1000 Mbps
- Full duplex and half duplex modes
 - IEEE 802.3x flow control automatic transmission of zero-quanta pause frame on flow control input deassertion
 - Optional forwarding of received pause control frames to the user
 - Packet bursting and frame extension in 1000 Mbps half-duplex
 - IEEE 802.3x flow control in full-duplex
 - Back-pressure support for half-duplex
- 4 KB TX FIFO RAM and 4 KB RX FIFO RAM with ECC support
- IEEE 1588-2002 and IEEE 1588-2008 precision networked clock synchronization
- IEEE 802.3-az, version D2.0 for Energy Efficient Ethernet (EEE)
- IEEE 802.1Q Virtual Local Area Network (VLAN) tag detection for reception frames
- Preamble and start-of-frame data (SFD) insertion in transmit and deletion in receive paths
- Automatic cyclic redundancy check (CRC) and pad generation controllable on a per-frame basis
- Options for automatic pad/CRC stripping on receive frames
- Programmable frame length supporting standard and jumbo Ethernet frames (with size up to 3800 bytes)
- Programmable inter-frame gap (IFG), from 40- to 96-bit times in steps of eight bits
- Preamble lengths of one, three, five and seven bytes supported
- Supports internal loopback asynchronous FIFO on the GMII/MII for debugging
- Supports a variety of flexible address filtering modes
 - Up to 31 additional 48-bit perfect destination address (DA) filters with masks for each byte
 - Up to 31 48-bit source address (SA) comparison check with masks for each byte
 - 256-bit hash filter (optional) for multicast and unicast DAs
 - Option to pass all multicast addressed frames
 - Promiscuous mode support to pass all frames without any filtering for network monitoring
 - Passes all incoming packets (as per filter) with a status report

DMA

- 32-bit interface
- Programmable burst size for optimal bus utilization
- Single-channel mode transmit and receive engines
- Byte-aligned addressing mode for data buffer support
- Dual-buffer (ring) or linked-list (chained) descriptor chaining
- Descriptors can each transfer up to 8 KB of data
- Independent DMA arbitration for transmit and receive with fixed priority or round robin

Management Interface

- 32-bit host interface to CSR set
- Comprehensive status reporting for normal operation and transfers with errors
- Configurable interrupt options for different operational conditions
- Per-frame transmit/receive complete interrupt control
- Separate status returned for transmission and reception packets

Acceleration

- Transmit and receive checksum offload for transmission control protocol (TCP), user datagram protocol (UDP), or Internet control message protocol (ICMP) over Internet protocol (IP)

PHY Interface

Different external PHY interfaces are provided depending on whether the Ethernet Controller signals are routed through the HPS I/O pins or the FPGA I/O pins.

The PHY interfaces supported using the HPS I/O pins are:

- Reduced Gigabit Media Independent Interface (RGMII)

The PHY interfaces supported using the FPGA I/O pins are:

- Media Independent Interface (MII)
- Gigabit Media Independent Interface (GMII)
- Reduced Media Independent Interface (RMII) with additional required adaptor logic

Note: Additional adaptor logic for RMII not provided.

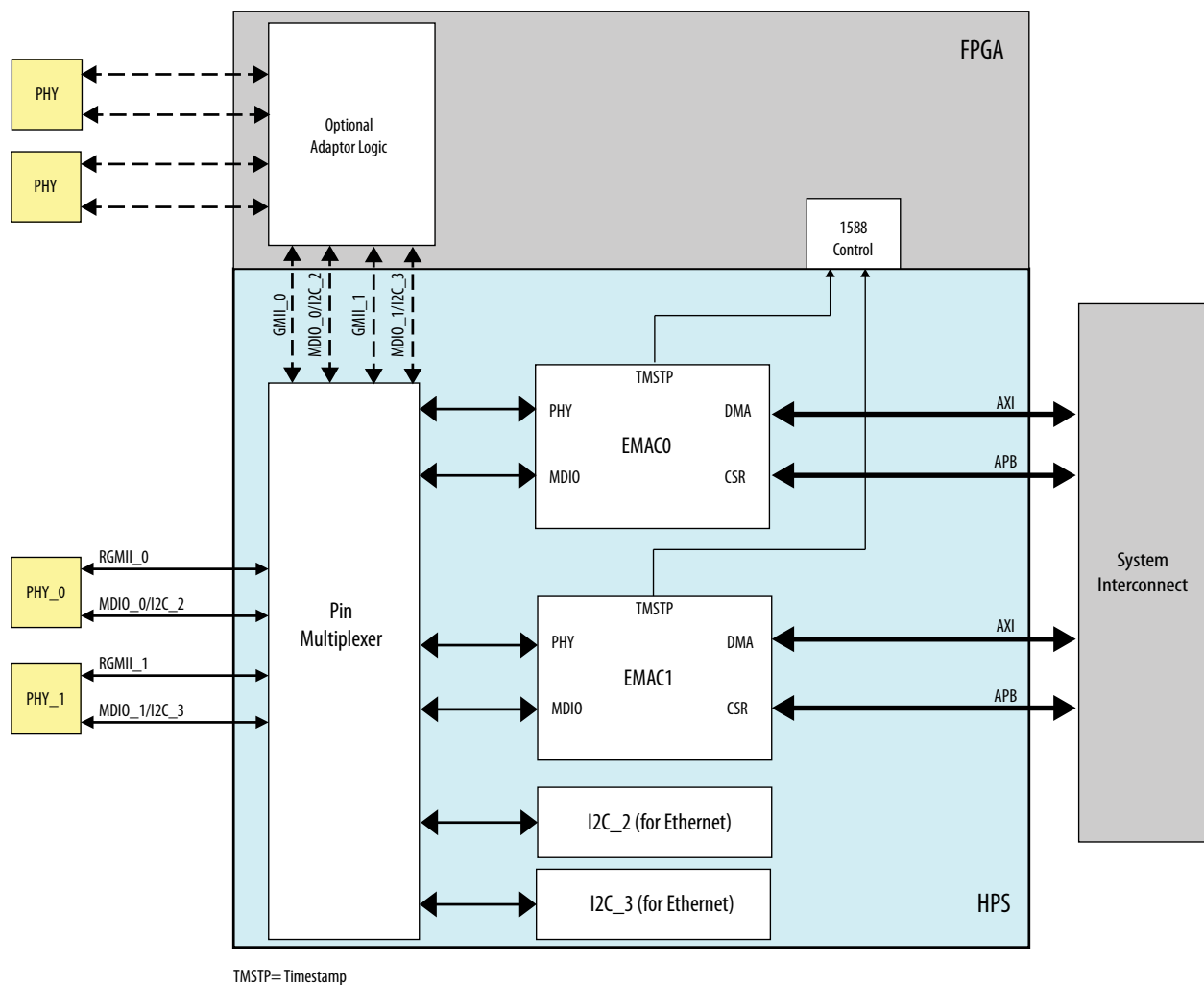
- Reduced Gigabit Media Independent Interface (RGMII) with additional required adaptor logic
- Serial Gigabit Media Independent Interface (SGMII) supported through transceiver I/O

The Ethernet Controller has two choices for the management control interface used for configuration and status monitoring of the PHY:

- Management Data Input/Output (MDIO)
- I²C PHY management through a separate I²C module within the HPS

EMAC Block Diagram and System Integration

Figure 17-1: EMAC System Integration



The EMACs are integrated into the HPS portion of the system on a chip (SoC) device. They communicate with the I/O pins.

EMAC Overview

Each EMAC is an internal bus master that sends Ethernet packets to and from the System Interconnect. The EMAC uses a descriptor ring protocol, where the descriptor contains an address to a buffer to fetch or store the packet data.

Each EMAC has an MDIO Management port to send commands to the external PHY. Alternatively, you can use an I²C module in the HPS for the management interface.

Each EMAC has an IEEE 1588 Timestamp interface with 20 ns resolution. The Arm Cortex-A9 microprocessor unit (MPU) subsystem can use it to maintain synchronization between the time counters that are internal to the three MACs. The clock reference for the timestamp can be provided by the Clock

Manager (emac_ptp_clk) or the FPGA fabric (f2s_emac_ptp_ref_clk). The clock reference is selected by the ptp_clk_sel bit in the emac_global register in the system manager.

EMAC Signal Description

The EMAC provides a variety of PHY interfaces and control options through the HPS and the FPGA I/Os.

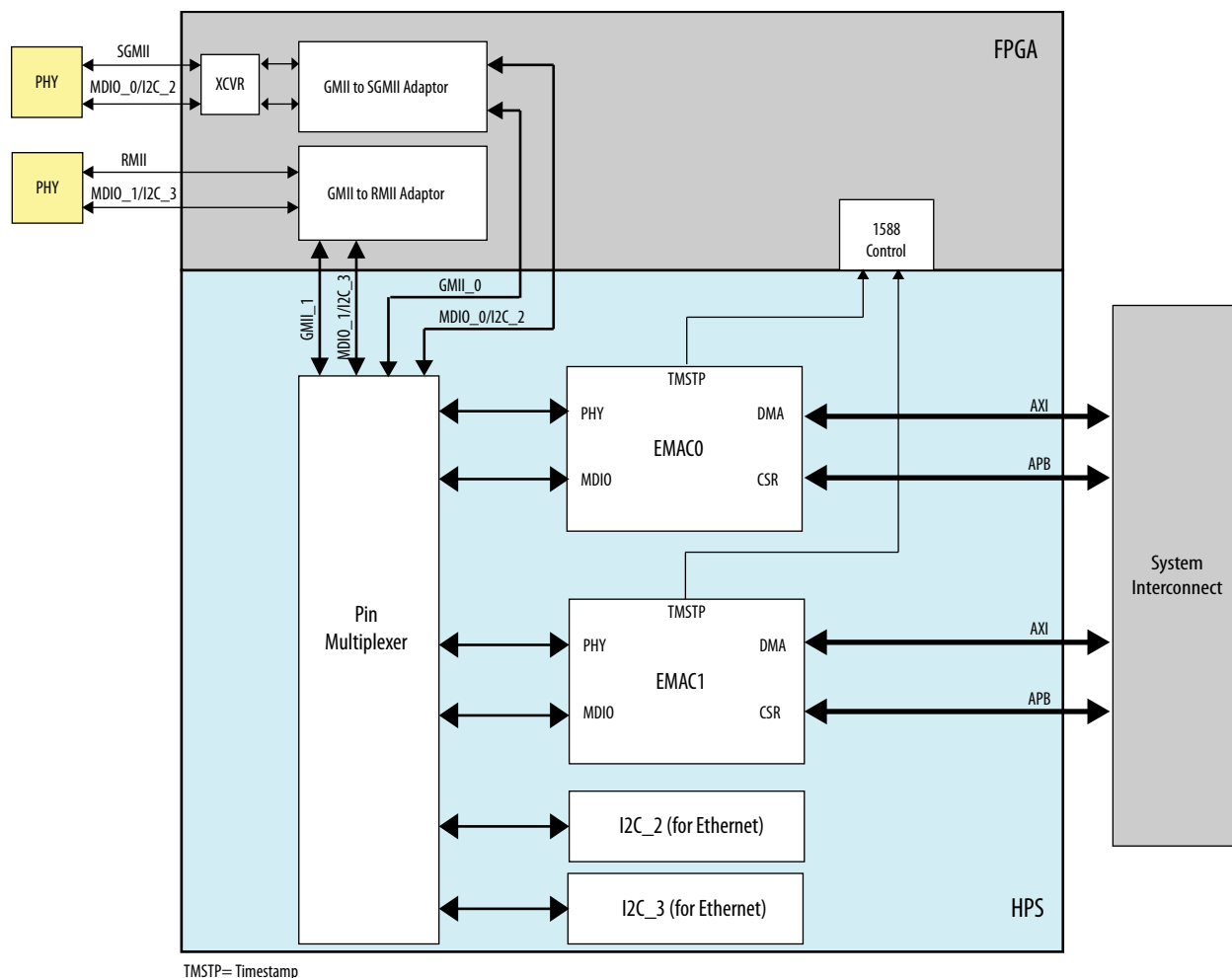
For designs in which the HPS is pin-limited, the EMAC signals can be routed through the FPGA as any one of the following interfaces by using soft adaptor logic in the FPGA:

- RMII/RGMII
- MII/GMII
- SGMII

The figure below depicts a design which routes the EMAC0 and EMAC1 signals through the FPGA to provide an RMII and SGMII interface.

Refer to the "EMAC FPGA Interface Initialization" section to find out more information about configuring EMAC interfaces through FPGA.

Figure 17-2: EMAC to FPGA Routing Example



Note: The Micrel KSZ9021RN Ethernet PHY has been verified to work properly with the HPS EMAC and is recommended for use in board design.

Related Information

[EMAC FPGA Interface Initialization](#) on page 17-65

Information on how to initialize GMII/MII interface

HPS EMAC I/O Signals

The following table lists the EMAC signals that are routed to the HPS pins. These signals provide the RGMII interface.

Table 17-1: HPS EMAC I/O Signals

EMAC Port		In/Out	Width	Description
phy_txclk_o	Transmit Clock	Out	1	This signal provides the transmit clock for RGMII (125/25/2.5 MHz in 1G/100M/10Mbps). All PHY transmit signals generated by the EMAC are synchronous to this clock.
phy_txd_o	PHY Transmit Data	Out	8	This group of eight transmit data signals is driven by the MAC. Bits [3:0] provide the RGMII transmit data. Unused bits in the RGMII interface configuration are tied low. In RGMII mode, the data bus carries transmit data at double rate and are sampled on both the rising and falling edges of the transmit clock. The validity of the data is qualified with phy_txen_o.
phy_txen_o	PHY Transmit Data Enable	Out	1	This signal is driven by the EMAC component, and in RGMII mode acts as the control signal (rgmii_tctl) for the transmit data, and is driven on both edges of the transmit clock, phy_txclk_o.
phy_clk_rx_i	Receive Clock	In	1	In RGMII mode, this clock frequency is 125/25/2.5 MHz in 1 G/100 M/10 Mbps modes. It is provided by the external PHY. All PHY signals received by the EMAC are synchronous to this clock.

EMAC Port		In/Out	Width	Description
phy_rxd_i	PHY Receive Data	In	8	These eight data signals are received from the PHY and carry receive data at double rate with bits[3:0] valid on the rising edge of phy_rxcclk_i, and bits[7:4] valid on the falling edge of phy_rxcclk_i. The validity of the data is qualified with phy_rxdv_i.
phy_rxdv_i	PHY Receive Data Valid	In	1	This signal is driven by the PHY and functions as the receive control signal used to qualify the data received on phy_rxd_i. This signal is sampled on both edges of the clock. Note that the signal phy_rxdv_i is assigned to pin RX_CTL in the device pin-out.

Note: The "n" in EMACn stands for the EMAC peripheral number.

Related Information

[EMAC HPS Interface Initialization](#) on page 17-66
Information on how to initialize RGMII interface

FPGA EMAC I/O Signals

Table 17-2: FPGA EMAC I/O Signals

Signal Name		In/Out	Width	Description
emac_clk_tx_i	Transmit Clock	In	1	This is the transmit clock (2.5 MHz/25 MHz) provided by the MII PHYs only. This clock comes from the FPGA Interface and is used for TX data capture. This clock is not used in GMII mode. Note: This clock must be able to perform glitch free switching between 2.5 and 25 MHz.

Signal Name		In/Out	Width	Description
emac_phy_txclk_o	Transmit Clock Output	Out	1	In GMII mode, this signal is the transmit clock output to the PHY to sample data. For MII, this clock is unused.
emac_phy_txd_o[7:0]	PHY Transmit Data	Out	8	These are a group of eight transmit data signals driven by the EMAC. All eight bits provide the GMII transmit data byte. For the lower speed MII operation, only bits[3:0] are used. The validity of the data is qualified with phy_txen_o and phy_txer_o. Synchronous to phy_txclk_o.
emac_phy_txen_o	PHY Transmit Data Enable	Out	1	This signal is driven by the EMAC and is used in GMII mode. When driven high, this signal indicates that valid data is being transmitted on the clk_tx_o bus.
emac_phy_txer_o	PHY Transmit Error	Out	1	This signal is driven by the EMAC and when high, indicates a transmit error or carrier extension on the phy_txd bus. It is also used to signal low power states in Energy Efficient Ethernet operation.
emac_rst_clk_tx_n_o	Transmit Clock Reset output	Out	1	Transmit clock reset output to the FPGA fabric, which is the internal synchronized reset to clk_tx_int output from the EMAC. May be used by logic implemented in the FPGA fabric as desired. The reset pulse width of the rst_clk_tx_n_o signal is three transmit clock cycles.

Signal Name		In/Out	Width	Description
emac_clk_rx_i	Receive Clock	In	1	Receive clock from external PHY. For GMII, the clock frequency is 125 MHz. For MII, the receive clock is 25 MHz for 100 Mbps and 2.5 MHz for 10 Mbps.
emac_phy_rxd_i[7:0]	PHY Receive Data	In	8	This is an eight-bit receive data bus from the PHY. In GMII mode, all eight bits are sampled. The validity of the data is qualified with phy_rxdv_i and phy_rxer_i. For lower speed MII operation, only bits [3:0] are sampled. These signals are synchronous to phy_clk_rx_i.
emac_phy_rxdv_i	PHY Receive Data Valid	In	1	This signal is driven by PHY. In GMII mode, when driven high, it indicates that the data on the phy_rxd bus is valid. It remains asserted continuously from the first recovered byte of the frame through the final recovered byte.
emac_phy_rxer_i	PHY Receive Error	In	1	This signal indicates an error or carrier extension (GMII) in the received frame. This signal is synchronous to phy_clk_rx_i.
emac_rst_clk_rx_n_o	Receive clock reset output.	Out	1	Receive clock reset output. The reset pulse width of the rst_clk_rx_n_o signal is three transmit clock cycles.



Signal Name		In/Out	Width	Description
emac_phy_crs_i	PHY Carrier Sense	In	1	This signal is asserted by the PHY when either the transmit or receive medium is not idle. The PHY de-asserts this signal when both transmit and receive interfaces are idle. This signal is not synchronous to any clock.
emac_phy_col_i	PHY Collision Detect	In	1	This signal, valid only when operating in half duplex, is asserted by the PHY when a collision is detected on the medium. This signal is not synchronous to any clock.

Related Information

[EMAC FPGA Interface Initialization](#) on page 17-65

Information on how to initialize GMII/MII interface

PHY Management Interface

The HPS can provide support for either MDIO or I²C PHY management interfaces.

MDIO Interface

The MDIO interface signals are synchronous to `l4_mp_clk` in all supported modes.

Note: The MDIO interface signals can be routed to both the FPGA and HPS I/O.

Table 17-3: PHY MDIO Management Interface

Signal	In/Out	Width	Description
emac_gmii_mdi_i	In	1	Management Data In. The PHY generates this signal to transfer register data during a read operation. This signal is driven synchronously with the <code>gmii_mdc_o</code> clock.
emac_gmii_mdo_o	Out	1	Management Data Out. The EMAC uses this signal to transfer control and data information to the PHY.

Signal	In/Out	Width	Description
emac_gmii_mdo_o_e	Out	1	Management Data Output Enable. This enable signal drives the gmii_mdo_o signal from an external three-state I/O buffer. This signal is asserted whenever valid data is driven on the gmii_mdo_o signal. The active state of this signal is high.
emac_gmii_mdc_o	Out	1	Management Data Clock. The EMAC provides timing reference for the gmii_md_i and gmii_mdo_o signals on MII through this aperiodic clock. The maximum frequency of this clock is 2.5 MHz. This clock is generated from the application clock through a clock divider.

I²C External PHY Management Interface

Some PHY devices use the I²C instead of MDIO for their control interface. Small form factor pluggable (SFP) optical or pluggable modules are often among those with this interface.

The HPS or FPGA can use two of the four general purpose I²C peripherals for controlling the PHY devices:

- I2C2 at base address 0xFFC06000
- I2C3 at base address 0xFFC07000

EMAC Internal Interfaces

DMA Master Interface

The DMA interface acts as a bus master on the system interconnect. Two types of data are transferred on the interface: data descriptors and actual data packets. The interface is very efficient in transferring full duplex Ethernet packet traffic. Read and write data transfers from different DMA channels can be performed simultaneously on this port, except for transmit descriptor reads and write-backs, which cannot happen simultaneously.

DMA transfers are split into a software configurable number of burst transactions on the interface. The AXI_Bus_Mode register in the dmagrp group is used to configure bursting behavior.

The interface assigns a unique ID for each DMA channel and also for each read DMA or write DMA request in a channel. Data transfers with distinct IDs can be reordered and interleaved.

The DMA interface can be configured to perform cacheable accesses. This configuration can be done in the System Manager when the DMA interface is inactive.

Write data transfers are generally performed as posted writes with OK responses returned as soon as the system interconnect has accepted the last beat of a data burst. Descriptors (status or timestamp), however, are always transferred as non-posted writes in order to prevent race conditions with the transfer complete interrupt logic.

The slave may issue an error response. When that happens, the EMAC disables the DMA channel that generated the original request and asserts an interrupt signal. The host must reset the EMAC with a hard or soft reset to restart the DMA to recover from this condition.

The EMAC supports up to 16 outstanding transactions on the interface. Buffering outstanding transactions smooths out back pressure behavior improving throughput when resource contention bottlenecks arise under high system load conditions.

Related Information

- [DMA Controller](#) on page 17-16
Information regarding DMA Controller functionality
- [System Manager](#) on page 6-1

Timestamp Interface

The timestamp clock reference can come from either the Clock Manager or the FPGA fabric. If the FPGA has implemented the serial capturing of the timestamp interface, then the FPGA must provide the PTP clock reference.

In addition to providing a timestamp clock reference, the FPGA can monitor the pulse-per-second output from each EMAC module and trigger a snapshot from each auxiliary time stamp timer.

The following table lists the EMAC to FPGA IEEE1588 Timestamp Interface signals to and from each EMAC module.

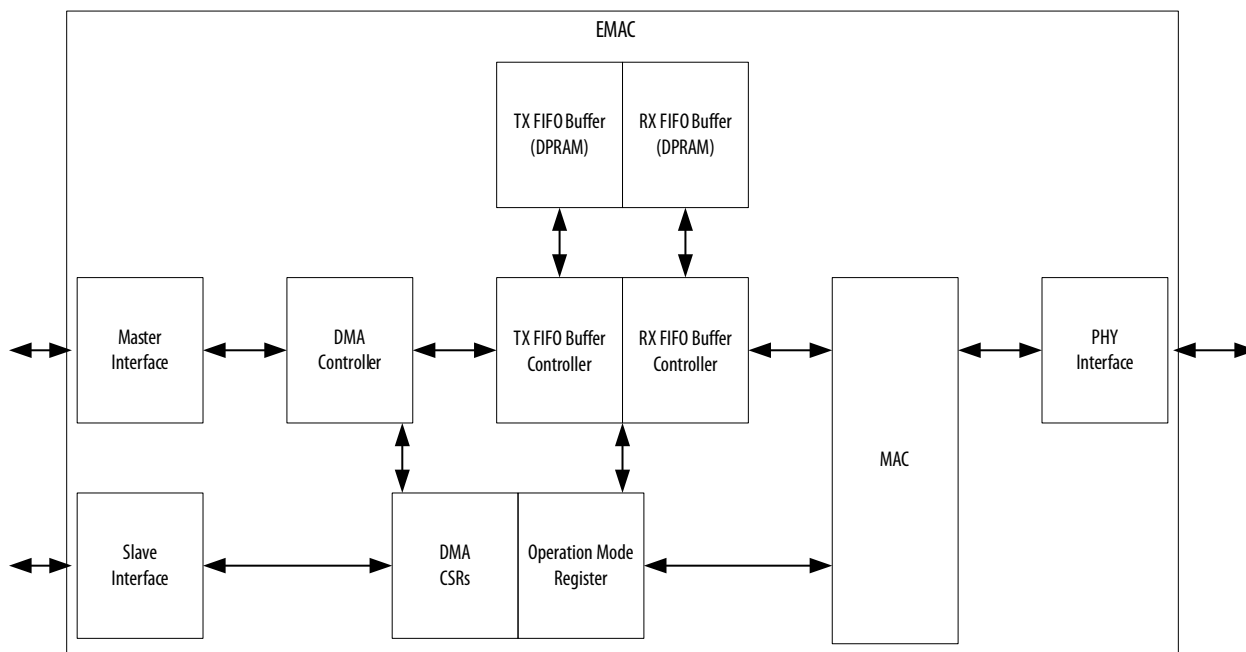
Table 17-4: EMAC to FPGA IEEE 1588 Timestamp Interface Signals

Signal Name		In/Out	Width	Description
f2h_emac_ptp_ref_clk	Timestamp PTP Clock reference from the FPGA	In	1	Used as PTP Clock reference for each EMAC when the FPGA has implemented Timestamp capture interface. Common for both EMACs.
ptp_pps_o	Pulse Per Second Output	Out	1	This signal is asserted based on the PPS mode selected in the Register 459 (PPS Control Register). Otherwise, this pulse signal is asserted every time the seconds counter is incremented. This signal is synchronous to f2h_emac_ptp_ref_clk and may only be sampled if the FPGA clock is used as timestamp reference.

Signal Name		In/Out	Width	Description
ptp_aux_ts_trig_i	Auxiliary Timestamp Trigger	In	1	This signal is asserted to take an auxiliary snapshot of the time. The rising edge of this internal signal is used to trigger the auxiliary snapshot. The signal is synchronized internally with <code>clk_ptp_ref_i</code> which results in an additional delay of 3 cycles. This input is asynchronous input and its assertion period must be greater than 2 PTP active clocks to be sampled.

Functional Description of the EMAC

Figure 17-3: EMAC High-Level Block Diagram with Interfaces



There are two host interfaces to the Ethernet MAC. The management host interface, a 32-bit slave interface, provides access to the CSR set. The data interface is a 32-bit master interface, and it controls data transfer between the direct memory access (DMA) controller channels and the rest of the HPS system through the system interconnect.

The built-in DMA controller is optimized for data transfer between the MAC controller and system memory. The DMA controller has independent transmit and receive engines and a CSR set. The transmit engine transfers data from system memory to the device port, while the receive engine transfers data from the device port to the system memory. The controller uses descriptors to efficiently move data from source to destination with minimal host intervention.

The EMAC also contains FIFO buffer memory to buffer and regulate the Ethernet frames between the application system memory and the EMAC module. Each EMAC module has one 4 KB TX FIFO and one 4 KB RX FIFO. On transmit, the Ethernet frames write into the transmit FIFO buffer, and eventually trigger the EMAC to perform the transfer. Received Ethernet frames are stored in the receive FIFO buffer and the FIFO buffer fill level is communicated to the DMA controller. The DMA controller then initiates the configured burst transfers. Receive and transmit transfer statuses are read by the EMAC and transferred to the DMA.

Transmit and Receive Data FIFO Buffers

Each EMAC component has associated transmit and receive data FIFO buffers to regulate the frames between the application system memory and the EMAC. Both FIFO buffer instances are 1024 x 42 bits. The FIFO buffer word consists of:

- Data: 32 bits
- Sideband:
 - Byte enables: 2 bits
 - End of frame (EOF): 1 bit
 - Error correction code (ECC): 7 bits

The data and sideband are protected by the 7-bit single error correct, double error detect (SECDED) code word.

The FIFO buffer RAMs have ECC enable, error injection and status pins. The enable and error injection pins are inputs driven by the system manager. The status pins are outputs driven to the MPU subsystem.

Note: The ECC block provides outputs to notify the system manager when single-bit correctable errors are detected (and corrected) and when double-bit uncorrectable errors are detected. The ECC logic also allows the injection of single-bit and double-bit errors for test purposes.

TX FIFO

The time at which data is sent from the TX FIFO to the EMAC is dependent on the transfer mode selected:

- Cut-through mode: Data is popped from the TX FIFO when the number of bytes in the TX FIFO crosses the configured threshold level (or when the end of the frame is written before the threshold is crossed). The threshold level is configured using the TTC bit of the Operation Mode Register (Register 6).

Note: After more than 96 bytes (or 548 bytes in 1000 Mbps mode) are popped to the EMAC, the TX FIFO controller frees that space and makes it available to the DMA and a retry is not possible.

- Store-and-Forward mode: Data is popped from the TX FIFO when one or more of the following conditions are true:
 - A complete frame is stored in the FIFO
 - The TX FIFO becomes almost full

The application can flush the TX FIFO of all contents by setting bit 20 (FTF) of Register 6 (Operation Mode Register). This bit is self-clearing and initializes the FIFO pointers to the default state. If the FTF bit is set during a frame transfer to the EMAC, further transfers are stopped because the FIFO is considered

empty. This cessation causes an underflow event and a runt frame to be transmitted and the corresponding status word is forwarded to the DMA.

If a collision occurs in half-duplex mode operation before an end of the frame, a retry attempt is sent before the end of the frame is transferred. When notified of the retransmission, the MAC pops the frame from the FIFO again.

Note: Only packets of 3800 bytes or less can be supported when the checksum offload feature is enabled by software.

RX FIFO

Frames received by the EMAC are pushed into the RX FIFO. The fill level of the RX FIFO is indicated to the DMA when it crosses the configured receive threshold which is programmed by the RTC field of Register 6 (Operation Mode Register). The time at which data is sent from the RX FIFO to the DMA is dependent on the configuration of the RX FIFO:

- Cut-through (default) mode: When 64 bytes or a full packet of data is received into the FIFO, data is popped out of the FIFO and sent to the DMA until a complete packet has been transferred. Upon completion of the end-of-frame transfer, the status word is popped and sent to the DMA.
- Store and forward mode: A frame is read out only after being written completely in the RX FIFO. This mode is configured by setting the RSF bit of Register 6 (Operation Mode Register).

If the RX FIFO is full before it receives the EOF data from the EMAC, an overflow is declared and the whole frame (including the status word) is dropped and the overflow counter in the DMA, (Register 8) Missed Frame and Buffer Overflow Counter Register, is incremented. This outcome is true even if the Forward Error Frame (FEF) bit of Register 6 (Operation Mode Register) is set. If the start address of such a frame has already been transferred, the rest of the frame is dropped and a dummy EOF is written to the FIFO along with the status word. The status indicates a partial frame because of overflow. In such frames, the Frame Length field is invalid. If the RX FIFO is configured to operate in the store-and-forward mode and if the length of the received frame is more than the FIFO size, overflow occurs and all such frames are dropped.

Note: In store-and-forward mode, only received frames with length 3800 bytes or less prevent overflow errors and frames from being dropped.

DMA Controller

The DMA has independent transmit and receive engines, and a CSR space. The transmit engine transfers data from system memory to the device port or MAC transaction layer (MTL), while the receive engine transfers data from the device port to the system memory. Descriptors are used to efficiently move data from source to destination with minimal Host CPU intervention. The DMA is designed for packet-oriented data transfers such as frames in Ethernet. The controller can be programmed to interrupt the Host CPU for situations such as frame transmit and receive transfer completion as well as error conditions.

The DMA and the Host driver communicate through two data structures:[†]

- Control and Status registers (CSR)[†]
- Descriptor lists and data buffers[†]

Because the Arria V implementation of the Ethernet MAC only supports enhanced descriptors, all mentions of descriptors in the following text refer to enhanced descriptors.

Descriptor Lists and Data Buffers[†]

The DMA transfers data frames received by the MAC to the receive Buffer in the Host memory, and transmit data frames from the transmit Buffer in the Host memory. Descriptors that reside in the Host memory act as pointers to these buffers.[†]

There are two descriptor lists: one for reception and one for transmission. The base address of each list is written into Register 3 (Receive Descriptor List Address Register) and Register 4 (Transmit Descriptor List Address Register), respectively. A descriptor list is forward linked (either implicitly or explicitly). The last descriptor may point back to the first entry to create a ring structure. Explicit chaining of descriptors is accomplished by setting the second address chained in both receive and transmit descriptors (RDES1[14] and TDES0[20]). The descriptor lists resides in the Host physical memory address space. Each descriptor can point to a maximum of two buffers. This enables two buffers to be used, physically addressed, rather than contiguous buffers in memory.[†]

A data buffer resides in the Host physical memory space, and consists of an entire frame or part of a frame, but cannot exceed a single frame. Buffers contain only data, buffer status is maintained in the descriptor. Data chaining refers to frames that span multiple data buffers. However, a single descriptor cannot span multiple frames. The DMA skips to the next frame buffer when end-of-frame is detected. Data chaining can be enabled or disabled.[†]

Figure 17-4: Descriptor Ring Structure

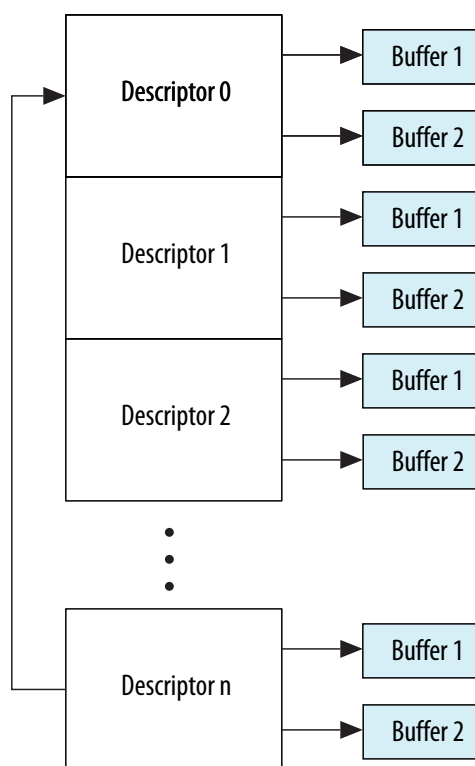
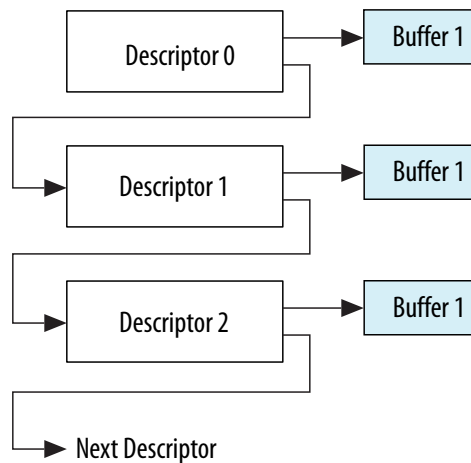


Figure 17-5: Descriptor Chain Structure



Note: You can select a descriptor structure during RTL configuration. The control bits in the descriptor structure are assigned so that the application can use an 8 KB buffer. All descriptions refer to the default descriptor structure.

Related Information

[Ethernet MAC Address Map and Register Definitions](#) on page 17-72

Information about control and status registers

Host Bus Burst Access

The DMA attempts to execute fixed-length burst transfers on the master interface if configured to do so through the FB bit of Register 0 (Bus Mode Register). The maximum burst length is indicated and limited by the PBL field (Bits [13:8]) Register 0 (Bus Mode Register). The receive and transmit descriptors are always accessed in the maximum possible (limited by packet burst length (PBL) or $16 * 8/\text{bus width}$) burst size for the 16- bytes to be read.

The transmit DMA initiates a data transfer only when the MTL transmit FIFO has sufficient space to accommodate the configured burst or the number of bytes remaining in the frame (when it is less than the configured burst length). The DMA indicates the start address and the number of transfers required to the master interface. When the interface is configured for fixed-length burst, it transfers data using the best combination of INCR4, 8, or 16 and SINGLE transactions. When not configured for fixed-length burst, it transfers data using INCR (undefined length) and SINGLE transactions.

The receive DMA initiates a data transfer only when sufficient data to accommodate the configured burst is available in MTL receive FIFO buffer or when the end of frame (when it is less than the configured burst length) is detected in the receive FIFO buffer. The DMA indicates the start address and the number of transfers required to the master interface. When the interface is configured for fixed-length burst, it transfers data using the best combination of INCR4, 8, or 16 and SINGLE transactions. If the end-of-frame is reached before the fixed burst ends on the interface, then dummy transfers are performed in order to complete the fixed burst. If the FB bit of Register 0 (Bus Mode Register) is clear, it transfers data using INCR (undefined length) and SINGLE transactions.

When the interface is configured for address aligned words, both DMA engines ensure that the first burst transfer initiated is less than or equal to the size of the configured packet burst length. Thus, all subsequent

beats start at an address that is aligned to the configured packet burst length. The DMA can only align the address for beats up to size 16 (for PBL > 16), because the interface does not support more than INCR16.

Host Data Buffer Alignment

The transmit and receive data buffers do not have any restrictions on start address alignment. For example, in systems with 32-bit memory, the start address for the buffers can be aligned to any of the four bytes. However, the DMA always initiates transfers with address aligned to the bus width with dummy data for the byte lanes not required. This typically happens during the transfer of the beginning or end of an Ethernet frame. The software driver should discard the dummy bytes based on the start address of the buffer and size of the frame.[†]

Example: Buffer Read

If the transmit buffer address is 0x00000FF2, and 15 bytes must be transferred, then the DMA reads five full words from address 0x00000FF0, but when transferring data to the MTL transmit FIFO buffer, the extra bytes (the first two bytes) are dropped or ignored. Similarly, the last three bytes of the last transfer are also ignored. The DMA always ensures it transfers data in 32-bit increments to the MTL transmit FIFO buffer, unless it is the end-of-frame.

Example: Buffer Write

If the receive buffer address is 0x00000FF2 and 16 bytes of a received frame must be transferred, then the DMA writes 3 full words from address 0x00000FF0. But the first two bytes of first transfer and the last two bytes of the fourth transfer have dummy data.

Buffer Size Calculations

The DMA does not update the size fields in the transmit and receive descriptors. The DMA updates only the status fields (RDES and TDES) of the descriptors. The driver must perform the size calculations.

The transmit DMA transfers the exact number of bytes (indicated by the buffer size field of TDES1) to the MAC. If a descriptor is marked as the first (FS bit of TDES1 is set), then the DMA marks the first transfer from the buffer as the start of frame. If a descriptor is marked as the last (LS bit of TDES1), then the DMA marks the last transfer from that data buffer as the end-of-frame to the MTL.

The receive DMA transfers data to a buffer until the buffer is full or the end-of-frame is received from the MTL. If a descriptor is not marked as the last (LS bit of RDES0), then the descriptor's corresponding buffer(s) are full and the amount of valid data in a buffer is accurately indicated by its buffer size field minus the data buffer pointer offset when the FS bit of that descriptor is set. The offset is zero when the data buffer pointer is aligned to the data bus width. If a descriptor is marked as the last, then the buffer may not be full (as indicated by the buffer size in RDES1). To compute the amount of valid data in this final buffer, the driver must read the frame length (FL bits of RDES0[29:16]) and subtract the sum of the buffer sizes of the preceding buffers in this frame. The receive DMA always transfers the start of next frame with a new descriptor.

Note: Even when the start address of a receive buffer is not aligned to the data width of system bus, the system should allocate a receive buffer of a size aligned to the system bus width. For example, if the system allocates a 1,024-byte (1 KB) receive buffer starting from address 0x1000, the software can program the buffer start address in the receive descriptor to have a 0x1002 offset. The receive DMA writes the frame to this buffer with dummy data in the first two locations (0x1000 and 0x1001). The actual frame is written from location 0x1002. Thus, the actual useful space in this buffer is 1,022 bytes, even though the buffer size is programmed as 1,024 bytes, because of the start address offset.

Transmission

The DMA can transmit with or without an optional second frame (OSF).

Related Information

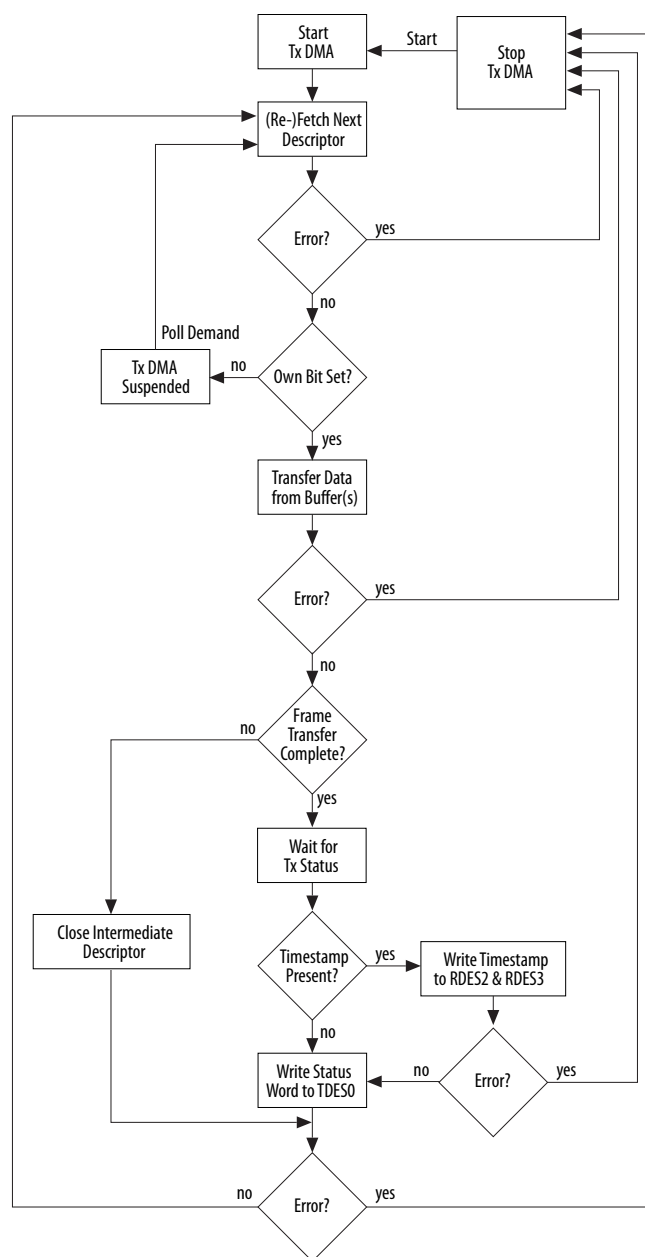
[Transmit Descriptor](#) on page 17-29

TX DMA Operation: Default (Non-OSF) Mode

The transmit DMA engine in default mode proceeds as follows: [†]

1. The Host sets up the transmit descriptor (TDES0-TDES3) and sets the Own bit (TDES0[31]) after setting up the corresponding data buffer(s) with Ethernet frame data. [†]
2. When Bit 13 (ST) of Register 6 (Operation Mode Register) is set, the DMA enters the Run state. [†]
3. While in the Run state, the DMA polls the transmit descriptor list for frames requiring transmission. After polling starts, it continues in either sequential descriptor ring order or chained order. If the DMA detects a descriptor flagged as owned by the Host (TDES0[31] = 0), or if an error condition occurs, transmission is suspended and both the Bit 2 (Transmit Buffer Unavailable) and Bit 16 (Normal Interrupt Summary) of the Register 5 (Status Register) are set. The transmit Engine proceeds to [step 9](#).
4. If the acquired descriptor is flagged as owned by DMA (TDES0[31] = 1), the DMA decodes the transmit Data Buffer address from the acquired descriptor.
5. The DMA fetches the transmit data from the Host memory and transfers the data to the MTL for transmission. [†]
6. If an Ethernet frame is stored over data buffers in multiple descriptors, the DMA closes the intermediate descriptor and fetches the next descriptor. Repeat [step 3](#), [step 4](#), and [step 5](#) until the end-of-Ethernet-frame data is transferred to the MTL. [†]
7. When frame transmission is complete, if IEEE 1588 timestamping was enabled for the frame (as indicated in the transmit status) the timestamp value obtained from MTL is written to the transmit descriptor (TDES2 and TDES3) that contains the end-of-frame buffer. The status information is then written to this transmit descriptor (TDES0). Because the Own bit is cleared during this step, the Host now owns this descriptor. If timestamping was not enabled for this frame, the DMA does not alter the contents of TDES2 and TDES3. [†]
8. Bit 0 (Transmit Interrupt) of Register 5 (Status Register) is set after completing transmission of a frame that has Interrupt on Completion (TDES1[31]) set in its Last descriptor. The DMA engine then returns to [step 3](#). [†]
9. In the Suspend state, the DMA tries to re-acquire the descriptor (and thereby return to [step 3](#)) when it receives a Transmit Poll demand and the Underflow Interrupt Status bit is cleared. [†]

Figure 17-6: TX DMA Operation in Default Mode

**TX DMA Operation: OSF Mode**

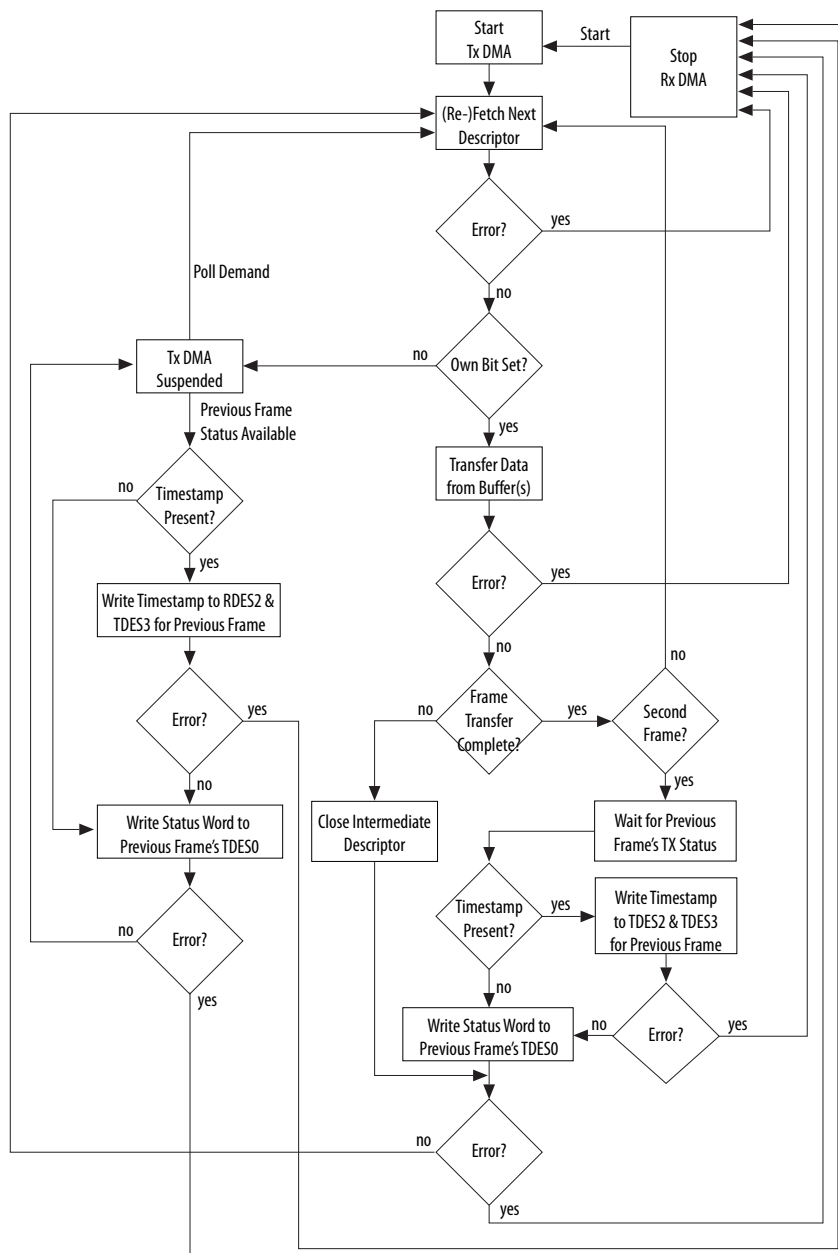
While in the Run state, the transmit process can simultaneously acquire two frames without closing the Status descriptor of the first [if Bit 2 (OSF) in Register 6 (Operation Mode Register) is set]. As the transmit process finishes transferring the first frame, it immediately polls the transmit descriptor list for the second frame. If the second frame is valid, the transmit process transfers this frame before writing the first frame's status information.[†]

In OSF mode, the Run state transmit DMA operates in the following sequence: [†]

1. The DMA operates as described in steps 1 - 6 of the [TX DMA Operation: Default \(Non-OSF\) Mode](#) on page 17-20 section.
2. Without closing the previous frame's last descriptor, the DMA fetches the next descriptor. [†]
3. If the DMA owns the acquired descriptor, the DMA decodes the transmit buffer address in this descriptor. If the DMA does not own the descriptor, the DMA goes into Suspend mode and skips to [step 7](#). [†]
4. The DMA fetches the transmit frame from the Host memory and transfers the frame to the MTL until the End-of-frame data is transferred, closing the intermediate descriptors if this frame is split across multiple descriptors. [†]
5. The DMA waits for the previous frame's frame transmission status and timestamp. Once the status is available, the DMA writes the timestamp to TDES2 and TDES3, if such timestamp was captured (as indicated by a status bit). The DMA then writes the status, with a cleared Own bit, to the corresponding TDES0, thus closing the descriptor. If timestamping was not enabled for the previous frame, the DMA does not alter the contents of TDES2 and TDES3. [†]
6. If enabled, the transmit interrupt is set, the DMA fetches the next descriptor, then proceeds to [step 3](#) (when Status is normal). If the previous transmission status shows an underflow error, the DMA goes into Suspend mode ([step 7](#)). [†]
7. In Suspend mode, if a pending status and timestamp are received from the MTL, the DMA writes the timestamp (if enabled for the current frame) to TDES2 and TDES3, then writes the status to the corresponding TDES0. It then sets relevant interrupts and returns to Suspend mode. [†]
8. The DMA can exit Suspend mode and enter the Run state (go to [step 1](#) or [step 2](#) depending on pending status) only after receiving a Transmit Poll demand (Register 1 (Transmit Poll Demand Register)). [†]

Note: As the DMA fetches the next descriptor in advance before closing the current descriptor, the descriptor chain should have more than two different descriptors for correct and proper operation. [†]

Figure 17-7: TX DMA Operation in OSF Mode



Transmit Frame Processing

The transmit DMA expects that the data buffers contain complete Ethernet frames, excluding preamble, pad bytes, and FCS fields and that the DA, SA, and Type/Len fields contain valid data. If the transmit descriptor indicates that the MAC must disable CRC or PAD insertion, the buffer must have complete Ethernet frames (excluding preamble), including the CRC bytes. †

Frames can be datachained and can span several buffers. Frames must be delimited by the First Descriptor (TDES1[29]) and the Last Descriptor (TDES1[30]), respectively. †

As transmission starts, the First Descriptor must have (TDES1[29]) set. When this occurs, frame data transfers from the Host buffer to the MTL transmit FIFO buffer. Concurrently, if the current frame has the Last Descriptor (TDES1[30]) clear, the transmit Process attempts to acquire the Next descriptor. The transmit Process expects this descriptor to have TDES1[29] clear. If TDES1[30] is clear, it indicates an intermediary buffer. If TDES1[30] is set, it indicates the last buffer of the frame. †

After the last buffer of the frame has been transmitted, the DMA writes back the final status information to the Transmit Descriptor 0 (TDES0) word of the descriptor that has the last segment set in Transmit Descriptor 1 (TDES1[30]). At this time, if Interrupt on Completion (TDES1[31]) is set, the Bit 0 (Transmit Interrupt) of Register 5 (Status Register) is set, the Next descriptor is fetched, and the process repeats. †

The actual frame transmission begins after the MTL transmit FIFO buffer has reached either a program-mable transmit threshold (Bits [16:14] of Register 6 (Operation Mode Register)), or a full frame is contained in the FIFO buffer. There is also an option for Store and Forward Mode (Bit 21 of Register 6 (Operation Mode Register)). Descriptors are released (Own bit TDES0[31] clears) when the DMA finishes transferring the frame. †

Note: To ensure proper transmission of a frame and the next frame, you must specify a non-zero buffer size for the transmit descriptor that has the Last Descriptor (TDES1[30]) set. †

Transmit Polling Suspended

Transmit polling can be suspended by either of the following conditions: †

- The DMA detects a descriptor owned by the Host (TDES0[31]=0). To resume, the driver must give descriptor ownership to the DMA and then issue a Poll Demand command. †
- A frame transmission is aborted when a transmit error because of underflow is detected. The appropriate Transmit Descriptor 0 (TDES0) bit is set. †

If the DMA goes into SUSPEND state because of the first condition, then both Bit 16 (Normal Interrupt Summary) and Bit 2 (Transmit Buffer Unavailable) of Register 5 (Status Register) are set. If the second condition occur, both Bit 15 (Abnormal Interrupt Summary) and Bit 5 (Transmit Underflow) of Register 5 (Status Register) are set, and the information is written to Transmit Descriptor 0, causing the suspension. †

In both cases, the position in the transmit List is retained. The retained position is that of the descriptor following the Last descriptor closed by the DMA. †

The driver must explicitly issue a Transmit Poll Demand command after rectifying the suspension cause. †

Reception

Receive functions use receive descriptors.

The receive DMA engine's reception sequence is proceeds as follows:

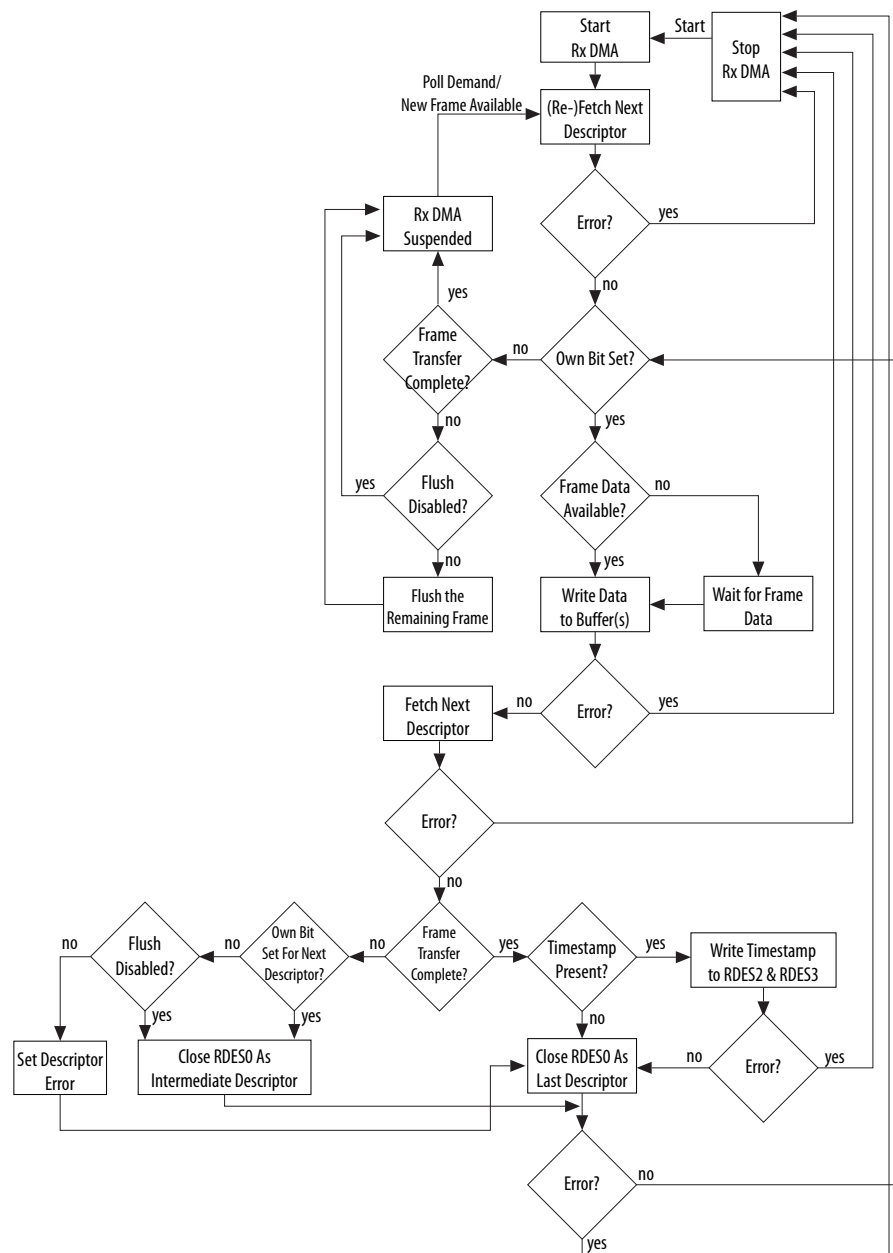
1. The host sets up receive descriptors (RDES0-RDES3) and sets the Own bit (RDES0[31]).†
2. When Bit 1 (SR) of Register 6 (Operation Mode Register) is set, the DMA enters the Run state. While in the Run state, the DMA polls the receive descriptor list, attempting to acquire free descriptors. If the fetched descriptor is not free (is owned by the host), the DMA enters the Suspend state and jumps to [step 9](#).†
3. The DMA decodes the receive data buffer address from the acquired descriptors.†
4. Incoming frames are processed and placed in the acquired descriptor's data buffers.†
5. When the buffer is full or the frame transfer is complete, the receive engine fetches the next descriptor.†
6. If the current frame transfer is complete, the DMA proceeds to [step 7](#). If the DMA does not own the next fetched descriptor and the frame transfer is not complete (EOF is not yet transferred), the DMA sets the Descriptor Error bit in the RDES0 (unless flushing is disabled in Bit 24 of Register 6 (Operation

Mode Register)). The DMA closes the current descriptor (clears the Own bit) and marks it as intermediate by clearing the Last Segment (LS) bit in the RDES0 value (marks it as Last Descriptor if flushing is not disabled), then proceeds to [step 8](#). If the DMA does own the next descriptor but the current frame transfer is not complete, the DMA closes the current descriptor as intermediate and reverts to [step 4](#).[†]

7. If IEEE 1588 timestamping is enabled, the DMA writes the timestamp (if available) to the current descriptor's RDES2 and RDES3. It then takes the receive frame's status from the MTL and writes the status word to the current descriptor's RDES0, with the Own bit cleared and the Last Segment bit set.[†]
8. The receive engine checks the latest descriptor's Own bit. If the host owns the descriptor (Own bit is 0), the Bit 7 (Receive Buffer Unavailable) of Register 5 (Status Register) is set and the DMA receive engine enters the Suspended state (Step 9). If the DMA owns the descriptor, the engine returns to [step 4](#) and awaits the next frame.
9. Before the receive engine enters the Suspend state, partial frames are flushed from the receive FIFO buffer. You can control flushing using Bit 24 of Register 6 (Operation Mode Register).[†]
10. The receive DMA exits the Suspend state when a Receive Poll demand is given or the start of next frame is available from the MTL's receive FIFO buffer. The engine proceeds to [step 2](#) and refetches the next descriptor.[†]



Figure 17-8: Receive DMA Operation



When software has enabled timestamping through the `t_sena` bit of register 448 (Timestamp Control Register) and a valid timestamp value is not available for the frame (for example, because the receive FIFO buffer was full before the timestamp could be written to it), the DMA writes all ones to RDES2 and RDES3 descriptors. Otherwise (that is, if timestamping is not enabled), the RDES2 and RDES3 descriptors remain unchanged.

Receive Descriptor Acquisition

The receive Engine always attempts to acquire an extra descriptor in anticipation of an incoming frame. Descriptor acquisition is attempted if any of the following conditions is satisfied: †

- Bit 1 (Start or Stop Receive) of Register 6 (Operation Mode Register) has been set immediately after being placed in the Run state. †
- The data buffer of the current descriptor is full before the frame ends for the current transfer. †
- The controller has completed frame reception, but the current receive descriptor is not yet closed. †
- The receive process has been suspended because of a host-owned buffer (RDES0[31] = 0) and a new frame is received. †
- A Receive poll demand has been issued. †

Receive Frame Processing

The MAC transfers the received frames to the Host memory only when the frame passes the address filter and frame size is greater than or equal to the configurable threshold bytes set for the receive FIFO buffer of MTL, or when the complete frame is written to the FIFO buffer in store-and-forward mode. †

If the frame fails the address filtering, it is dropped in the MAC block itself (unless Bit 31 (Receive All) of Register 1 (MAC Frame Filter) is set). Frames that are shorter than 64 bytes, because of collision or premature termination, can be removed from the MTL receive FIFO buffer. †

After 64 (configurable threshold) bytes have been received, the MTL block requests the DMA block to begin transferring the frame data to the receive buffer pointed by the current descriptor. The DMA sets the First Descriptor (RDES0[9]) after the DMA Host interface becomes ready to receive a data transfer (if the DMA is not fetching transmit data from the host), to delimit the frame. The descriptors are released when the Own (RDES[31]) bit is clear, either as the Data buffer fills up or as the last segment of the frame is transferred to the receive buffer. If the frame is contained in a single descriptor, both Last Descriptor (RDES0[8]) and First Descriptor (RDES0[9]) are set.

The DMA fetches the next descriptor, sets the Last Descriptor (RDES[8]) bit, and releases the RDES0 status bits in the previous frame descriptor. Then the DMA sets bit 6 (Receive Interrupt) of Register 5 (Status Register). The same process repeats unless the DMA encounters a descriptor flagged as being owned by the host. If this occurs, Bit 7 (Receive Buffer Unavailable) of Register 5 (Status Register) is set and the receive process enters the Suspend state. The position in the receive list is retained. †

Receive Process Suspended

If a new receive frame arrives while the receive process is in the suspend state, the DMA refetches the current descriptor in the Host memory. If the descriptor is now owned by the DMA, the receive process re-enters the run state and starts frame reception. If the descriptor is still owned by the host, by default, the DMA discards the current frame at the top of the MTL RX FIFO buffer and increments the missed frame counter. If more than one frame is stored in the MTL EX FIFO buffer, the process repeats. †

The discarding or flushing of the frame at the top of the MTL EX FIFO buffer can be avoided by disabling Flushing (Bit 24 of Register 6 (Operation Mode Register)). In such conditions, the receive process sets the Receive Buffer Unavailable status and returns to the Suspend state. †

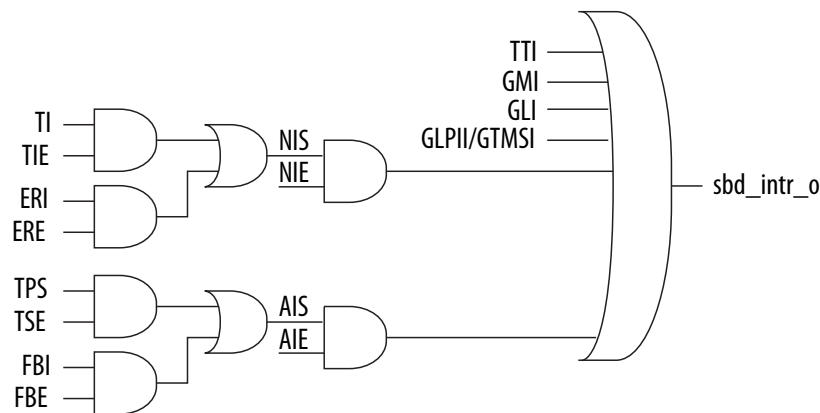
Interrupts

Interrupts can be generated as a result of various events. The DMA Register 5 (Status Register) contains a status bit for each of the events that can cause an interrupt. Register 7 (Interrupt Enable Register) contains an enable bit for each of the possible interrupt sources.

There are two groups of interrupts, Normal and Abnormal, as described in Register 5 (Status Register). Interrupts are cleared by writing a 1 to the corresponding bit position. When all the enabled interrupts

within a group are cleared, the corresponding summary bit is cleared. When both the summary bits are cleared, the `sbd_intr_o` interrupt signal is deasserted. If the MAC is the cause for assertion of the interrupt, then any of the GLI, GMI, TTI, or GLPII bits of Register 5 (Status Register) are set to 1.

Figure 17-9: Summary Interrupt (`sbd_intr_o`) Generation⁽⁵³⁾



Note: Register 5 (Status Register) is the interrupt status register. The interrupt pin (`sbd_intr_o`) is asserted because of any event in this status register only if the corresponding interrupt enable bit is set in Register 7 (Interrupt Enable Register).[†]

Interrupts are not queued, and if the interrupt event occurs before the driver has responded to it, no additional interrupts are generated. For example, Bit 6 (Receive Interrupt) of Register 5 (Status Register) indicates that one or more frames were transferred to the Host buffer. The driver must scan all descriptors, from the last recorded position to the first one owned by the DMA.

An interrupt is generated only once for multiple, simultaneous events. The driver must scan Register 5 (Status Register) for the cause of the interrupt. After the driver has cleared the appropriate bit in Register 5 (Status Register), the interrupt is not generated again until a new interrupting event occurs. For example, the controller sets Bit 6 (Receive Interrupt) of Register 5 (Status Register) and the driver begins reading Register 5 (Status Register). Next, the interrupt indicated by Bit 7 (Receive Buffer Unavailable) of Register 5 (Status Register) occurs. The driver clears the receive interrupt (bit 6). However, the `sbd_intr_o` signal is not deasserted, because of the active or pending Receive Buffer Unavailable interrupt.

Bits 7:0 (`riwt` field) of Register 9 (Receive Interrupt Watchdog Timer Register) provide for flexible control of the receive interrupt. When this Interrupt timer is programmed with a non-zero value, it gets activated as soon as the RX DMA completes a transfer of a received frame to system memory without asserting the receive Interrupt because it is not enabled in the corresponding Receive Descriptor (`RDES1[31]`). When this timer runs out as per the programmed value, the AIS bit is set and the interrupt is asserted if the corresponding AIE is enabled in Register 7 (Interrupt Enable Register). This timer is disabled before it runs out, when a frame is transferred to memory, and the receive interrupt is triggered if it is enabled.

Related Information

[Receive Descriptor](#) on page 17-36

Error Response to DMA

If the slave replies with an error response to any data transfer initiated by a DMA channel, that DMA stops all operations and updates the error bits and the Fatal Bus Error bit in the Register 5 (Status Register). The

⁽⁵³⁾ Signals NIS and AIS are registered.

DMA controller can resume operation only after soft resetting or hard resetting the EMAC and reinitializing the DMA.

Descriptor Overview

The DMA in the Ethernet subsystem transfers data based on a single enhanced descriptor, as explained in the DMA Controller section. The enhanced descriptor is created in the system memory. The descriptor addresses must be word-aligned.

The enhanced or alternate descriptor format can have 8 DWORDS (32 bytes) instead of 4 DWORDS as in the case of the normal descriptor format.

The features of the enhanced or alternate descriptor structure are:

- The alternative descriptor structure is implemented to support buffers of up to 8 KB (useful for Jumbo frames).[†]
- There is a re-assignment of control and status bits in TDES0, TDES1, RDES0 (advanced timestamp or IPC full offload configuration), and RDES1.[†]
- The transmit descriptor stores the timestamp in TDES6 and TDES7 when you select the advanced timestamp.[†]
- The receive descriptor structure is also used for storing the extended status (RDES4) and timestamp (RDES6 and RDES7) when advanced timestamp, IPC Full Checksum Offload Engine, or Layer 3 and Layer 4 filter feature is selected.[†]
- You can select one of the following options for descriptor structure:
 - If timestamping is enabled in Register 448 (Timestamp Control Register) or Checksum Offload is enabled in Register 0 (MAC Configuration Register), the software must to allocate 32 bytes (8 DWORDS) of memory for every descriptor by setting Bit 7 (Descriptor Size) of Register 0 (Bus Mode Register).
 - If timestamping or Checksum Offload is not enabled, the extended descriptors (DES4 to DES7) are not required. Therefore, software can use descriptors with the default size of 16 bytes (4 DWORDS) by clearing Bit 7 (Descriptor Size) of Register 0 (Bus Mode Register) to 0.

Related Information

[DMA Controller](#) on page 17-16

Transmit Descriptor

The application software must program the control bits TDES0[31:18] during the transmit descriptor initialization. When the DMA updates the descriptor, it writes back all the control bits except the OWN bit (which it clears) and updates the status bits[7:0].

With the advance timestamp support, the snapshot of the timestamp to be taken can be enabled for a given frame by setting Bit 25 (TTSE) of TDES0. When the descriptor is closed (that is, when the OWN bit is cleared), the timestamp is written into TDES6 and TDES7 as indicated by the status Bit 17 (TTSS) of TDES0.

Note: Only enhanced descriptor formats (4 or 8 DWORDS) are supported.

Note: When the advanced timestamp feature is enabled, software should set Bit 7 of Register 0 (Bus Mode Register), so that the DMA operates with extended descriptor size. When this control bit is clear, the TDES4-TDES7 descriptor space is not valid.[†]

Figure 17-10: Transmit Enhanced Descriptor Fields - Format

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TDES0	OWN	Ctrl [30:26]					T T S E	Ctrl [24:18]					T T S S	Status [16:7]							Ctrl/Status [6:3]			Status [2:0]								
TDES1	RES		Buffer 2 Byte Count [28:16]												RES		Buffer 1 Byte Count [12:0]															
TDES2	Buffer 1 Address [31:0]																															
TDES3	Buffer 2 Address [31:0] or Next Descriptor Address [31:0]																															
TDES4	Reserved																															
TDES5	Reserved																															
TDES6	Transmit Timestamp Low [31:0]																															
TDES7	Transmit Timestamp High [31:0]																															

The DMA always reads or fetches four DWORDS of the descriptor from system memory to obtain the buffer and control information. [†]

Figure 17-11: Transmit Descriptor Fetch (Read) Format

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TDES0	OWN	Ctrl [30:26]					T T S E	Ctrl [24:18]					Reserved for Status [17:7]										SLOT Number [6:3]				Reserved for Status [2:0]					
TDES1	RES [31:29]		Buffer 2 Byte Count [28:16]										RES		Buffer 1 Byte Count [12:0]																	
TDES2	Buffer 1 Address [31:0]																															
TDES3	Buffer 2 Address [31:0] or Next Descriptor Address [31:0]																															

Table 17-5: Transmit Descriptor Word 0 (TDES0)

Bit	Description
31	OWN: Own Bit When set, this bit indicates that the descriptor is owned by the DMA. When this bit is cleared, it indicates that the descriptor is owned by the Host. The DMA clears this bit either when it completes the frame transmission or when the buffers allocated in the descriptor are read completely. The ownership bit of the frame's first descriptor must be set after all subsequent descriptors belonging to the same frame have been set to avoid a possible race condition between fetching a descriptor and the driver setting an ownership bit. [†]
30	IC: Interrupt on Completion When set, this bit enables the Transmit Interrupt (Register 5[0]) to be set after the present frame has been transmitted. [†]
29	LS: Last Segment When set, this bit indicates that the buffer contains the last segment of the frame. When this bit is set, the TBS1 or TBS2 field in TDES1 should have a non-zero value. [†]
28	FS: First Segment When set, this bit indicates that the buffer contains the first segment of a frame. [†]
27	DC: Disable CRC When this bit is set, the MAC does not append a CRC to the end of the transmitted frame. This bit is valid only when the first segment (TDES0[28]) is set. [†]

Bit	Description
26	DP: Disable Pad When set, the MAC does not automatically add padding to a frame shorter than 64 bytes. When this bit is cleared, the DMA automatically adds padding and CRC to a frame shorter than 64 bytes, and the CRC field is added despite the state of the DC (TDES0[27]) bit. This bit is valid only when the first segment (TDES0[28]) is set. [†]
25	TTSE: Transmit Timestamp Enable When set, this bit enables IEEE1588 hardware timestamping for the transmit frame referenced by the descriptor. This field is valid only when the First Segment control bit (TDES0[28]) is set.
24	Reserved
23:22	CIC: Checksum Insertion Control. These bits control the checksum calculation and insertion. The following list describes the bit encoding: ▪ 0x0: Checksum insertion disabled. ▪ 0x1: Only IP header checksum calculation and insertion are enabled. ▪ 0x2: IP header checksum and payload checksum calculation and insertion are enabled, but pseudoheader checksum is not calculated in hardware. ▪ 0x3: IP Header checksum and payload checksum calculation and insertion are enabled, and pseudoheader checksum is calculated in hardware. This field is valid when the First Segment control bit (TDES0[28]) is set.
21	TER: Transmit End of Ring When set, this bit indicates that the descriptor list reached its final descriptor. The DMA returns to the base address of the list, creating a descriptor ring. [†]
20	TCH: Second Address Chained When set, this bit indicates that the second address in the descriptor is the Next descriptor address rather than the second buffer address. When TDES0[20] is set, TBS2 (TDES1[28:16]) is a “don’t care” value. TDES0[21] takes precedence over TDES0[20]. [†]
19:18	Reserved
17	TTSS: Transmit Timestamp Status This field is used as a status bit to indicate that a timestamp was captured for the described transmit frame. When this bit is set, TDES2 and TDES3 have a timestamp value captured for the transmit frame. This field is only valid when the descriptor’s Last Segment control bit (TDES0[29]) is set. [†]

Bit	Description
16	IHE: IP Header Error When set, this bit indicates that the MAC transmitter detected an error in the IP datagram header. The transmitter checks the header length in the IPv4 packet against the number of header bytes received from the application and indicates an error status if there is a mismatch. For IPv6 frames, a header error is reported if the main header length is not 40 bytes. Furthermore, the Ethernet Length/Type field value for an IPv4 or IPv6 frame must match the IPheader version received with the packet. For IPv4 frames, an error status is also indicated if the Header Length field has a value less than 0x5. [†] This bit is valid only when the Tx Checksum Offload is enabled. If COE detects an IP header error, it still inserts an IPv4 header checksum if the Ethernet Type field indicates an IPv4 payload. [†]
15	ES: Error Summary Indicates the logical OR of the following bits: ▪ TDES0[14]: Jabber Timeout ▪ TDES0[13]: Frame Flush ▪ TDES0[11]: Loss of Carrier ▪ TDES0[10]: No Carrier ▪ TDES0[9]: Late Collision ▪ TDES0[8]: Excessive Collision ▪ TDES0[2]: Excessive Deferral ▪ TDES0[1]: Underflow Error ▪ TDES0[16]: IP Header Error ▪ TDES0[12]: IP Payload Error [†]
14	JT: Jabber Timeout When set, this bit indicates the MAC transmitter has experienced a jabber time-out. This bit is only set when Bit 22 (Jabber Disable) of Register 0 (MAC Configuration Register) is not set. [†]
13	FF: Frame Flushed When set, this bit indicates that the DMA or MTL flushed the frame because of a software Flush command given by the CPU. [†]
12	IPE: IP Payload Error When set, this bit indicates that MAC transmitter detected an error in the TCP, UDP, or ICMP IP datagram payload. The transmitter checks the payload length received in the IPv4 or IPv6 header against the actual number of TCP, UDP, or ICMP packet bytes received from the application and issues an error status in case of a mismatch. [†]

Bit	Description
11	LC: Loss of Carrier When set, this bit indicates that a loss of carrier occurred during frame transmission (that is, the gmii_crs_i signal was inactive for one or more transmit clock periods during frame transmission). This bit is valid only for the frames transmitted without collision when the MAC operates in the half-duplex mode. [†]
10	NC: No Carrier When set, this bit indicates that the Carrier Sense signal from the PHY was not asserted during transmission. [†]
9	LC: Late Collision When set, this bit indicates that frame transmission is aborted because of a collision occurring after the collision window (64 byte-times, including preamble, in MII mode and 512 byte-times, including preamble and carrier extension, in GMII mode). This bit is not valid if the Underflow Error bit is set.
8	EC: Excessive Collision When set, this bit indicates that the transmission was aborted after 16 successive collisions while attempting to transmit the current frame. If Bit 9 (Disable Retry) in Register 0 (MAC Configuration Register) is set, this bit is set after the first collision, and the transmission of the frame is aborted. [†]
7	VF: VLAN Frame When set, this bit indicates that the transmitted frame is a VLAN-type frame. [†]
6:3	CC: Collision Count (Status field) These status bits indicate the number of collisions that occurred before the frame was transmitted. This count is not valid when the Excessive Collisions bit (TDES0[8]) is set. The EMAC updates this status field only in the half-duplex mode.
2	ED: Excessive Deferral When set, this bit indicates that the transmission has ended because of excessive deferral of over 24,288 bit times (155,680 bits times in 1,000-Mbps mode or if Jumbo frame is enabled) if Bit 4 (Deferral Check) bit in Register 0 (MAC Configuration Register) is set. [†]
1	UF: Underflow Error When set, this bit indicates that the MAC aborted the frame because the data arrived late from the Host memory. Underflow Error indicates that the DMA encountered an empty transmit buffer while transmitting the frame. The transmission process enters the Suspended state and sets both Transmit Underflow (Register 5[5]) and Transmit Interrupt (Register 5[0]). [†]
0	DB: Deferred Bit When set, this bit indicates that the MAC defers before transmission because of the presence of carrier. This bit is valid only in the half-duplex mode. [†]

Table 17-6: Transmit Descriptor Word 1 (TDES1)

Bit	Description
31:29	Reserved
28:16	TBS2: Transmit Buffer 2 Size This field indicates the second data buffer size in bytes. This field is not valid if TDES0[20] is set. [†]
15:13	Reserved [†]
12:0	TBS1: Transmit Buffer 1 Size This field indicates the first data buffer byte size, in bytes. If this field is 0, the DMA ignores this buffer and uses Buffer 2 or the next descriptor, depending on the value of TCH (TDES0[20]). [†]

Table 17-7: Transmit Descriptor 2 (TDES2)

Bit	Description
31:0	Buffer 1 Address Pointer These bits indicate the physical address of Buffer 1. There is no limitation on the buffer address alignment. [†]

Table 17-8: Transmit Descriptor 3 (TDES3)

Bit	Description
31:0	Buffer 2 Address Pointer (Next Descriptor Address) Indicates the physical address of Buffer 2 when a descriptor ring structure is used. If the Second Address Chained (TDES0[20]) bit is set, this address contains the pointer to the physical memory where the Next descriptor is present. The buffer address pointer must be aligned to the bus width only when TDES0[20] is set. (LSBs are ignored internally.) [†]

Table 17-9: Transmit Descriptor 6 (TDES6)

Bit	Description
31:0	TTSL: Transmit Frame Timestamp Low This field is updated by DMA with the least significant 32 bits of the timestamp captured for the corresponding transmit frame. This field has the timestamp only if the Last Segment bit (LS) in the descriptor is set and Timestamp status (TTSS) bit is set. [†]

Table 17-10: Transmit Descriptor 7 (TDES7)

Bit	Description
31:0	TTSH: Transmit Frame Timestamp High This field is updated by DMA with the most significant 32 bits of the timestamp captured for the corresponding receive frame. This field has the timestamp only if the Last Segment bit (LS) in the descriptor is set and Timestamp status (TTSS) bit is set. [†]

Receive Descriptor

The receive descriptor can have 32 bytes of descriptor data (8 DWORDs) when advanced timestamp or IPC Full Offload feature is selected. When either of these features is enabled, software should set bit 7 of Register 0 (Bus Mode Register) so that the DMA operates with extended descriptor size. When this control bit is clear, the RDES0[0] is always cleared and the RDES4-RDES7 descriptor space is not valid. [†]

Note: Only enhanced descriptor formats (4 or 8 DWORDS) are supported.

Figure 17-12: Receive Enhanced Descriptor Fields Format

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RDES0	O W N	Status [30:0]																														
RDES1	C T R L	R E S [30:29]		Buffer 2 Byte Count [28:16]												C t r l [15:14]		R E S	Buffer 1 Byte Count [12:0]													
RDES2	Buffer 1 Address [31:0]																															
RDES3	Buffer 2 Address [31:0] or Next Descriptor Address [31:0]																															
RDES4	Extended status [31:0]																															
RDES5	Reserved																															
RDES6	Receive Timestamp Low [31:0]																															
RDES7	Receive Timestamp High[31:0]																															

Receive Descriptor Field 0 (RDES0)

Table 17-11: Receive Descriptor Field 0 (RDES0)

Bit	Description
31	OWN: Own Bit When set, this bit indicates that the descriptor is owned by the DMA of the EMAC. When this bit is cleared, this bit indicates that the descriptor is owned by the Host. The DMA clears this bit either when it completes the frame reception or when the buffers that are associated with this descriptor are full.
30	AFM: Destination Address Filter Fail When set, this bit indicates a frame that failed in the DA Filter in the MAC. [†]
29:16	FL: Frame Length These bits indicate the byte length of the received frame that was transferred to host memory (including CRC). This field is valid when Last Descriptor (RDES0[8]) is set and either the Descriptor Error (RDES0[14]) or Overflow Error bits are cleared. The frame length also includes the two bytes appended to the Ethernet frame when IP checksum calculation (Type 1) is enabled and the received frame is not a MAC control frame. This field is valid when Last Descriptor (RDES0[8]) is set. When the Last Descriptor and Error Summary bits are not set, this field indicates the accumulated number of bytes that have been transferred for the current frame. [†]

Bit	Description
15	ES: Error Summary Indicates the logical OR of the following bits: • RDES0[1]: CRC Error • RDES0[3]: Receive Error • RDES0[4]: Watchdog Timeout • RDES0[6]: Late Collision • RDES0[7]: Giant Frame • RDES4[4:3]: IP Header or Payload Error (Receive Descriptor Field 4 (RDES4)) • RDES0[11]: Overflow Error • RDES0[14]: Descriptor Error This field is valid only when the Last Descriptor (RDES0[8]) is set. [†]
14	DE: Descriptor Error When set, this bit indicates a frame truncation caused by a frame that does not fit within the current descriptor buffers, and that the DMA does not own the Next descriptor. The frame is truncated. This bit is valid only when the Last Descriptor (RDES0[8]) bit is set. [†]
13	SAF: Source Address Filter Fail When set, this bit indicates that the SA field of frame failed the SA Filter in the MAC. [†]
12	LE: Length Error When set, this bit indicates that the actual length of the frame received and that the Length/ Type field does not match. This bit is valid only when the Frame Type (RDES0[5]) bit is clear. [†]
11	OE: Overflow Error When set, this bit indicates that the received frame was damaged because of buffer overflow in MTL. Note: This bit is set only when the DMA transfers a partial frame to the application, which happens only when the RX FIFO buffer is operating in the threshold mode. In the store-and-forward mode, all partial frames are dropped completely in the RX FIFO buffer. [†]

Bit	Description
10	VLAN: VLAN Tag When set, this bit indicates that the frame to which this descriptor is pointing is a VLAN frame tagged by the MAC. The VLAN tagging depends on checking the VLAN fields of the received frame based on the Register 7 (VLAN Tag Register) setting. †
9	FS: First Descriptor When set, this bit indicates that this descriptor contains the first buffer of the frame. If the size of the first buffer is 0, the second buffer contains the beginning of the frame. If the size of the second buffer is also 0, the next descriptor contains the beginning of the frame. †
8	LD: Last Descriptor When set, this bit indicates that the buffers pointed to by this descriptor are the last buffers of the frame. †
7	Timestamp Available When set, bit[7] indicates that a snapshot of the Timestamp is written in descriptor words 6 (RDES6) and 7 (RDES7). This is valid only when the Last Descriptor bit (RDES0[8]) is set.
6	LC: Late Collision When set, this bit indicates that a late collision has occurred while receiving the frame in the half-duplex mode. †
5	FT: Frame Type When set, this bit indicates that the receive frame is an Ethernet-type frame (the LT field is greater than or equal to 0x0600). When this bit is cleared, it indicates that the received frame is an IEEE802.3 frame. This bit is not valid for Runt frames less than 14 bytes.
4	RWT: Receive Watchdog Timeout When set, this bit indicates that the receive Watchdog Timer has expired while receiving the current frame and the current frame is truncated after the Watchdog Timeout. †



Bit	Description
3	RE: Receive Error When set, this bit indicates that the <code>gmii_rxr_i</code> signal is asserted while <code>gmii_rxdv_i</code> is asserted during frame reception.
2	DE: Dribble Bit Error When set, this bit indicates that the received frame has a non-integer multiple of bytes (odd nibbles). This bit is valid only in the MII Mode. [†]
1	CE: CRC Error When set, this bit indicates that a CRC error occurred on the received frame. This bit is valid only when the Last Descriptor (RDES0[8]) is set. [†]
0	Extended Status Available/RX MAC Address When either advanced timestamp or IP Checksum Offload (Type 2) is present, this bit, when set, indicates that the extended status is available in descriptor word 4 (RDES4). This bit is valid only when the Last Descriptor bit (RDES0[8]) is set. When the Advance Timestamp Feature or IPC Full Offload is not selected, this bit indicates RX MAC Address status. When set, this bit indicates that the RX MAC Address registers value (1 to 15) matched the frame's DA field. When clear, this bit indicates that the RX MAC Address Register 0 value matched the DA field. [†]

Related Information

- [Receive Descriptor Field 4 \(RDES4\)](#) on page 17-43
- [Receive Descriptor Field 6 \(RDES6\)](#) on page 17-45
- [Receive Descriptor Field 7 \(RDES7\)](#) on page 17-46

Receive Descriptor Field 1 (RDES1)

Table 17-12: Receive Descriptor Field 1 (RDES1)

Bit	Description
31	DIC: Disable Interrupt on Completion When set, this bit prevents setting the Status Register's RI bit (CSR5[6]) for the received frame ending in the buffer indicated by this descriptor. As a result, the RI interrupt for the frame is disabled and is not asserted to the Host. [†]
30:29	Reserved [†]
28:16	RBS2: Receive Buffer 2 Size These bits indicate the second data buffer size, in bytes. The buffer size must be a multiple of 4, even if the value of RDES3 (buffer2 address pointer) in the Receive Descriptor Field 3 (RDES3) is not aligned to the bus width. If the buffer size is not an appropriate multiple of 4, the resulting behavior is undefined. This field is not valid if RDES1[14] is set. For more information about calculating buffer sizes, refer to the Buffer Size Calculations section in this chapter.
15	RER: Receive End of Ring When set, this bit indicates that the descriptor list reached its final descriptor. The DMA returns to the base address of the list, creating a descriptor ring. [†]
14	RCH: Second Address Chained When set, this bit indicates that the second address in the descriptor is the next descriptor address rather than the second buffer address. When this bit is set, RBS2 (RDES1[28:16]) is a “don't care” value. RDES1[15] takes precedence over RDES1[14]. [†]
13	Reserved [†]
12:0	RBS1: Receive Buffer 1 Size Indicates the first data buffer size in bytes. The buffer size must be a multiple of 4, even if the value of RDES2 (buffer1 address pointer), in the Receive Descriptor Field 2 (RDES2), is not aligned. When the buffer size is not a multiple of 4, the resulting behavior is undefined. If this field is 0, the DMA ignores this buffer and uses Buffer 2 or the next descriptor depending on the value of RCH (Bit 14). For more information about calculating buffer sizes, refer to the Buffer Size Calculations section in this chapter.

Related Information

- [Buffer Size Calculations](#) on page 17-19
- [Receive Descriptor Field 2 \(RDES2\)](#) on page 17-42
- [Receive Descriptor Field 3 \(RDES3\)](#) on page 17-42

Receive Descriptor Fields (RDES2) and (RDES3)

Receive Descriptor Field 2 (RDES2)

Table 17-13: Receive Descriptor Field 2 (RDES2)

Bit	Description
31:0	Buffer 1 Address Pointer These bits indicate the physical address of Buffer 1. There are no limitations on the buffer address alignment except for the following condition: The DMA uses the value programmed in RDES2[1:0] for its address generation when the RDES2 value is used to store the start of the frame. The DMA performs a write operation with the RDES2[1:0] bits as 0 during the transfer of the start of the frame but the frame is shifted as per the actual buffer address pointer. The DMA ignores RDES2[1:0] if the address pointer is to a buffer where the middle or last part of the frame is stored. For more information about buffer address alignment, refer to the <i>Host Data Buffer Alignment</i> section.

Related Information

[Host Data Buffer Alignment](#) on page 17-19

Receive Descriptor Field 3 (RDES3)

Table 17-14: Receive Descriptor Field 3 (RDES3)

Bit	Description
31:0	Buffer 2 Address Pointer (Next Descriptor Address) These bits indicate the physical address of Buffer 2 when a descriptor ring structure is used. If the Second Address Chained (RDES1[14]) bit in Receive Descriptor Field 1 (RDES1) is set, this address contains the pointer to the physical memory where the Next descriptor is present. If RDES1[14], in the Receive Descriptor Field 1 (RDES1) is set, the buffer (Next descriptor) address pointer must be bus width-aligned (RDES3[1:0] = 0. LSBs are ignored internally.) However, when RDES1[14] in the Receive Descriptor Field 1 (RDES1) is cleared, there are no limitations on the RDES3 value, except for the following condition: the DMA uses the value programmed in RDES3 [1:0] for its buffer address generation when the RDES3 value is used to store the start of frame. The DMA ignores RDES3 [1:0] if the address pointer is to a buffer where the middle or last part of the frame is stored.

Related Information

- [Host Data Buffer Alignment](#) on page 17-19
- [Receive Descriptor Field 1 \(RDES1\)](#) on page 17-41

Receive Descriptor Field 4 (RDES4)

The extended status is written only when there is status related to IPC or timestamp available. The availability of extended status is indicated by Bit 0 in RDES0. This status is available only when the Advance Timestamp or IPC Full Offload feature is selected.

Table 17-15: Receive Descriptor Field 4 (RDES4)

Bit	Description
31:28	Reserved [†]
27:26	Layer 3 and Layer 4 Filter Number Matched These bits indicate the number of the Layer 3 and Layer 4 Filter that matched the received frame. • 00: Filter 0 • 01: Filter 1 • 10: Filter 2 • 11: Filter 3 This field is valid only when Bit 24 or Bit 25 is set. When more than one filter matches, these bits give only the lowest filter number. [†]
25	Layer 4 Filter Match When set, this bit indicates that the received frame matches one of the enabled Layer 4 Port Number fields. This status is given only when one of the following conditions is true: • Layer 3 fields are not enabled and all enabled Layer 4 fields match. • All enabled Layer 3 and Layer 4 filter fields match. When more than one filter matches, this bit gives the layer 4 filter status of filter indicated by Bits [27:26]. [†]
24	Layer 3 Filter Match When set, this bit indicates that the received frame matches one of the enabled Layer 3 IP Address fields. This status is given only when one of the following conditions is true: • All enabled Layer 3 fields match and all enabled Layer 4 fields are bypassed. • All enabled filter fields match. When more than one filter matches, this bit gives the layer 3 filter status of the filter indicated by Bits [27:26]. [†]
23:15	Reserved
14	Timestamp Dropped When set, this bit indicates that the timestamp was captured for this frame but got dropped in the MTL RX FIFO buffer because of overflow.

Bit	Description
13	PTP Version When set, this bit indicates that the received PTP message has the IEEE 1588 version 2 format. When clear, it has the version 1 format.
12	PTP Frame Type When set, this bit indicates that the PTP message is sent directly over Ethernet. When this bit is not set and the message type is non-zero, it indicates that the PTP message is sent over UDP-IPv4 or UDP-IPv6. The information about IPv4 or IPv6 can be obtained from Bits 6 and 7.
11:8	Message Type These bits are encoded to give the type of the message received. • 0000: No PTP message received • 0001: SYNC (all clock types) • 0010: Follow_Up (all clock types) • 0011: Delay_Req (all clock types) • 0100: Delay_Resp (all clock types) • 0101: Pdelay_Req (in peer-to-peer transparent clock) • 0110: Pdelay_Resp (in peer-to-peer transparent clock) • 0111: Pdelay_Resp_Follow_Up (in peer-to-peer transparent clock) • 1000: Announce • 1001: Management • 1010: Signaling • 1011-1110: Reserved • 1111: PTP packet with Reserved message type
7	IPv6 Packet Received When set, this bit indicates that the received packet is an IPv6 packet. This bit is updated only when Bit 10 (IPC) of Register 0 (MAC Configuration Register) is set.
6	IPv4 Packet Received When set, this bit indicates that the received packet is an IPv4 packet. This bit is updated only when Bit 10 (IPC) of Register 0 (MAC Configuration Register) is set.
5	IP Checksum Bypassed When set, this bit indicates that the checksum offload engine is bypassed.
4	IP Payload Error When set, this bit indicates that the 16-bit IP payload checksum (that is, the TCP, UDP, or ICMP checksum) that the EMAC calculated does not match the corresponding checksum field in the received segment. It is also set when the TCP, UDP, or ICMP segment length does not match the payload length value in the IP Header field. This bit is valid when either Bit 7 or Bit 6 is set.

Bit	Description
3	IP Header Error When set, this bit indicates that either the 16-bit IPv4 header checksum calculated by the EMAC does not match the received checksum bytes, or the IP datagram version is not consistent with the Ethernet Type value. This bit is valid when either Bit 7 or Bit 6 is set.
2:0	IP Payload Type These bits indicate the type of payload encapsulated in the IP datagram processed by the receive Checksum Offload Engine (COE). The COE also sets these bits to 0 if it does not process the IP datagram's payload due to an IP header error or fragmented IP. • 0x0: Unknown or did not process IP payload • 0x1: UDP • 0x2: TCP • 0x3: ICMP • 0x4–0x7: Reserved This bit is valid when either Bit 7 or Bit 6 is set.

Related Information[Receive Descriptor Field 0 \(RDES0\)](#) on page 17-37**Receive Descriptor Fields (RDES6) and (RDES7)**

Receive Descriptor Fields 6 and 7 (RDES6 and RDES7) contain the snapshot of the timestamp. The availability of the snapshot of the timestamp in RDES6 and RDES7 is indicated by Bit 7 in the RDES0 descriptor.

Related Information[Receive Descriptor Field 0 \(RDES0\)](#) on page 17-37**Receive Descriptor Field 6 (RDES6)****Table 17-16: Receive Descriptor Field 6 (RDES6)**

Bit	Description
31:0	RTSL: Receive Frame Timestamp Low This field is updated by the DMA with the least significant 32 bits of the timestamp captured for the corresponding receive frame. This field is updated by the DMA only for the last descriptor of the receive frame, which is indicated by Last Descriptor status bit (RDES0[8]) in RDES0.

Related Information[Receive Descriptor Field 0 \(RDES0\)](#) on page 17-37

Receive Descriptor Field 7 (RDES7)

Table 17-17: Receive Descriptor Field 7 (RDES7)

Bit	Description
31:0	RTSH: Receive Frame Timestamp High This field is updated by the DMA with the most significant 32 bits of the timestamp captured for the corresponding receive frame. This field is updated by the DMA only for the last descriptor of the receive frame, which is indicated by Last Descriptor status bit (RDES0[8]) in RDES0.

Related Information

[Receive Descriptor Field 0 \(RDES0\)](#) on page 17-37

IEEE 1588-2002 Timestamps

The IEEE 1588-2002 standard defines the Precision Time Protocol (PTP) that enables precise synchronization of clocks in a distributed network of devices. The PTP applies to systems communicating by local area networks supporting multicast messaging. This protocol enables heterogeneous systems that include clocks of varying inherent precision, resolution, and stability to synchronize. It is frequently used in automation systems where a collection of communicating machines such as robots must be synchronized and hence operate over a common time base.^(†)

The PTP is transported over UDP/IP. The system or network is classified into Master and Slave nodes for distributing the timing and clock information.[†]

The following figure shows the process that PTP uses for synchronizing a slave node to a master node by exchanging PTP messages.

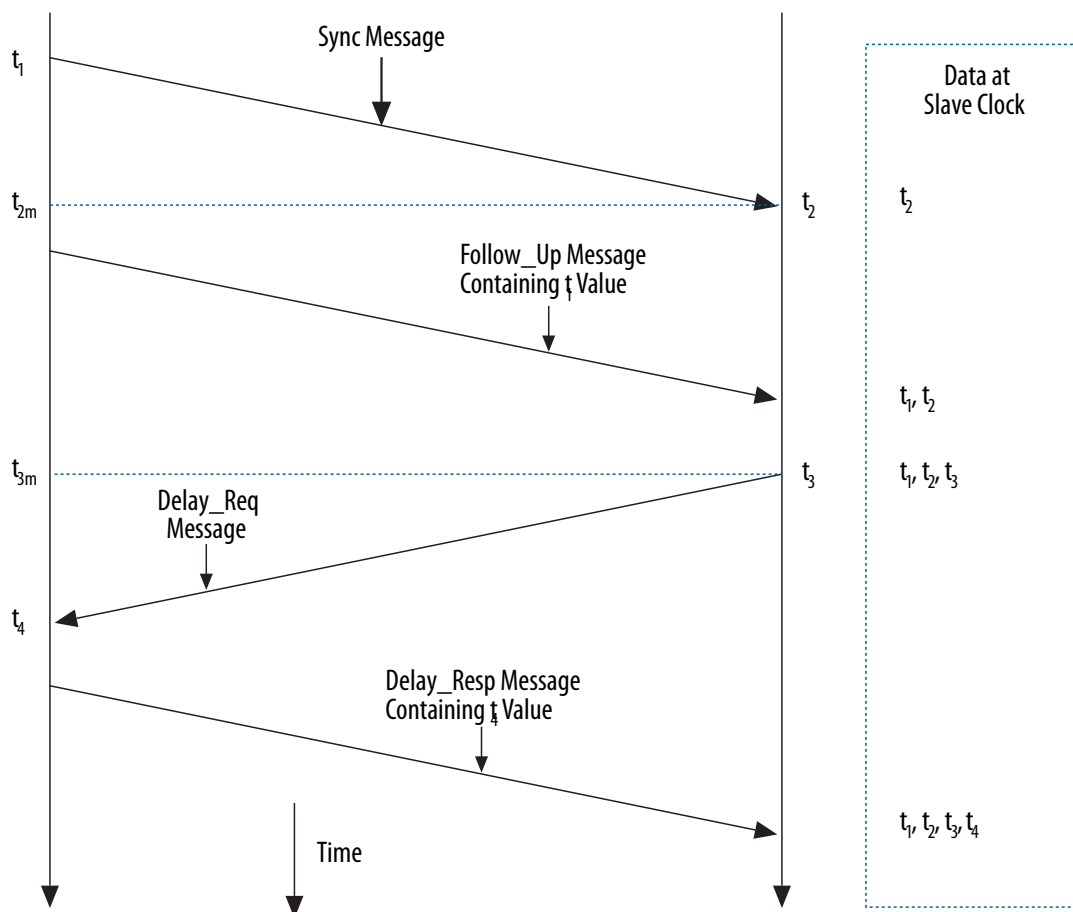
^(†) Portions © 2017 Synopsys, Inc. Used with permission. All rights reserved. Synopsys & DesignWare are registered trademarks of Synopsys, Inc. All documentation is provided "as is" and without any warranty. Synopsys expressly disclaims any and all warranties, express, implied, or otherwise, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, and any warranties arising out of a course of dealing or usage of trade.

[†]Paragraphs marked with the dagger (†) symbol are Synopsys Proprietary. Used with permission.

Figure 17-13: Networked Time Synchronization

Master Clock Time

Slave Clock Time



The PTP uses the following process for synchronizing a slave node to a master node by exchanging the PTP messages:

1. The master broadcasts the PTP Sync messages to all its nodes. The Sync message contains the master's reference time information. The time at which this message leaves the master's system is t_1 . This time must be captured, for Ethernet ports, at the PHY interface.[†]
2. The slave receives the sync message and also captures the exact time, t_2 , using its timing reference.[†]
3. The master sends a follow_up message to the slave, which contains t_1 information for later use.[†]
4. The slave sends a delay_req message to the master, noting the exact time, t_3 , at which this frame leaves the PHY interface.[†]
5. The master receives the message, capturing the exact time, t_4 , at which it enters its system.[†]
6. The master sends the t_4 information to the slave in the delay_resp message.[†]
7. The slave uses the four values of t_1 , t_2 , t_3 , and t_4 to synchronize its local timing reference to the master's timing reference.[†]

Most of the PTP implementation is done in the software above the UDP layer. However, the hardware support is required to capture the exact time when specific PTP packets enter or leave the Ethernet port at

the PHY interface. This timing information must be captured and returned to the software for the proper implementation of PTP with high accuracy.[†]

The EMAC is intended to support IEEE 1588 operation in all modes with a resolution of 20 ns. When the two EMACs are operating in an IEEE 1588 environment, the MPU subsystem is responsible for maintaining synchronization between the time counters internal to the two MACs.

The IEEE 1588 interface to the FPGA allows the FPGA to provide a source for the `emac_ptp_ref_clk` input as well to allow it to monitor the pulse per second output from each EMAC controller.

The EMAC component provides a hardware assisted implementation of the IEEE 1588 protocol. Hardware support is for timestamp maintenance. Timestamps are updated when receiving any frame on the PHY interface, and the receive descriptor is updated with this value. Timestamps are also updated when the SFD of a frame is transmitted and the transmit descriptor is updated accordingly.[†]

Note: You may use external time-stamp clock reference to accomplish timing. Use the `set_global_assignment -name ENABLE_HPS_INTERNAL_TIMING ON` to enable timing analysis.

Related Information

IEEE Standards Association

For details about the IEEE 1588-2002 standard, refer to IEEE Standard 1588-2002 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, available on the IEEE Standards Association website.[†]

Reference Timing Source

To get a snapshot of the time, the EMAC takes the reference clock input and uses it to generate the reference time (64-bit) internally and capture timestamps.[†]

System Time Register Module

The 64-bit time is maintained in this module and updated using the input reference clock, `clk_ptp_ref`, which can be the `emac_ptp_clk` from the HPS or the `f2s_ptp_ref_clk` from the FPGA. The `emac_ptp_clk` in the HPS is a derivative of the `osc1_clk` and is configured in the clock manager. This input reference clock is the source for taking snapshots (timestamps) of Ethernet frames being transmitted or received at the PHY interface.

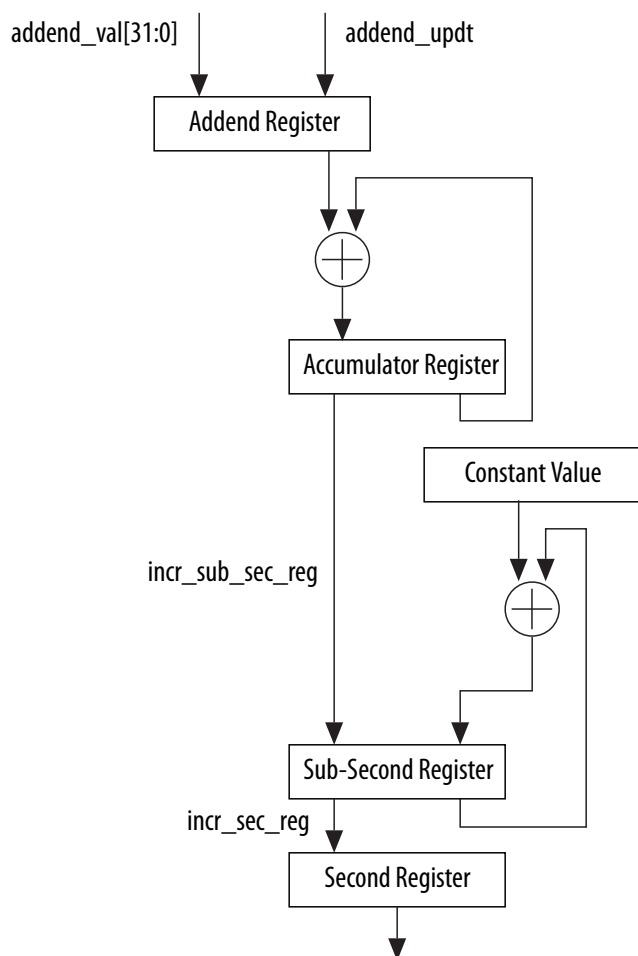
The system time counter can be initialized or corrected using the coarse correction method. In this method, the initial value or the offset value is written to the Timestamp Update register. For initialization, each EMAC's system time counter is written with the value in the Timestamp Update registers, while for system time correction, the offset value is added to or subtracted from the system time.

With the fine correction method, a slave clock's frequency drift with respect to the master clock is corrected over a period of time instead of in one clock, as in coarse correction. This protocol helps maintain linear time and does not introduce drastic changes (or a large jitter) in the reference time between PTP sync message intervals.[†]

With this method, an accumulator sums up the contents of the Timestamp_Addend register, as shown in the figure below. The arithmetic carry that the accumulator generates is used as a pulse to increment the system time counter. The accumulator and the addend are 32-bit registers. Here, the accumulator acts as a high-precision frequency multiplier or divider.

Note: You must connect a PTP clock with a frequency higher than the frequency required for the specified accuracy.[†]

Figure 17-14: Algorithm for System Time Update Using Fine Method



The System Time Update logic requires a 50-MHz clock frequency to achieve 20-ns accuracy. The frequency division ratio (FreqDivisionRatio) is the ratio of the reference clock frequency to the required clock frequency. Hence, if the reference clock (clk_ptp_ref_i) is for example, 66 MHz, this ratio is calculated as $66 \text{ MHz} / 50 \text{ MHz} = 1.32$. Hence, the default addend value to program in the register is $2^{32} / 1.32$, 0xC1F07C1F.

If the reference clock drifts lower, to 65 MHz for example, the ratio is $65 / 50$, or 1.3 and the value to set in the addend register is $2^{32} / 1.30$, or 0xC4EC4EC4. If the clock drifts higher, to 67 MHz for example, the addend register must be set to 0xBF0B7672. When the clock drift is nil, the default addend value of 0xC1F07C1F ($2^{32} / 1.32$) must be programmed.[†]

In the above figure, the constant value used to accumulate the sub-second register is decimal 43, which achieves an accuracy of 20 ns in the system time (in other words, it is incremented in 20-ns steps).

The software must calculate the drift in frequency based on the Sync messages and update the Addend register accordingly.[†]

Initially, the slave clock is set with `FreqCompensationValue0` in the Addend register. This value is as follows:[†]

$$\text{FreqCompensationValue}_0 = 2^{32} / \text{FreqDivisionRatio}^{\dagger}$$

If `MasterToSlaveDelay` is initially assumed to be the same for consecutive sync messages, the algorithm described below must be applied. After a few sync cycles, frequency lock occurs. The slave clock can then determine a precise `MasterToSlaveDelay` value and re-synchronize with the master using the new value.[†]

The algorithm is as follows:[†]

- At time `MasterSyncTimen` the master sends the slave clock a sync message. The slave receives this message when its local clock is `SlaveClockTimen` and computes `MasterClockTimen` as:[†]

$$\text{MasterClockTime}_n = \text{MasterSyncTime}_n + \text{MasterToSlaveDelay}_n^{\dagger}$$

- The master clock count for current sync cycle, `MasterClockCountn` is given by:[†]

$$\text{MasterClockCount}_n = \text{MasterClockTime}_n - \text{MasterClockTime}_{n-1}$$

(assuming that `MasterToSlaveDelay` is the same for sync cycles n and $n - 1$)[†]

- The slave clock count for current sync cycle, `SlaveClockCountn` is given by:[†]

$$\text{SlaveClockCount}_n = \text{SlaveClockTime}_n - \text{SlaveClockTime}_{n-1}^{\dagger}$$

- The difference between master and slave clock counts for current sync cycle, `ClockDiffCountn` is given by:[†]

$$\text{ClockDiffCount}_n = \text{MasterClockCount}_n - \text{SlaveClockCount}_n^{\dagger}$$

- The frequency-scaling factor for the slave clock, `FreqScaleFactorn` is given by:[†]

$$\text{FreqScaleFactor}_n = (\text{MasterClockCount}_n + \text{ClockDiffCount}_n) / \text{SlaveClockCount}_n^{\dagger}$$

- The frequency compensation value for Addend register, `FreqCompensationValuen` is given by:[†]

$$\text{FreqCompensationValue}_n = \text{FreqScaleFactor}_n \times \text{FreqCompensationValue}_{n-1} - 1^{\dagger}$$

In theory, this algorithm achieves lock in one Sync cycle; however, it may take several cycles, because of changing network propagation delays and operating conditions.[†]

This algorithm is self-correcting; if for any reason the slave clock is initially set to a value from the master that is incorrect, the algorithm corrects it at the cost of more Sync cycles.[†]

Transmit Path Functions

The MAC captures a timestamp when the start-of-frame data (SFD) is sent on the PHY interface. You can control the frames for which timestamps are captured on a per frame basis. In other words, each transmit frame can be marked to indicate whether a timestamp should be captured for that frame.[†]

You can use the control bits in the transmit descriptor to indicate whether a timestamp should be capture for a frame. The MAC returns the timestamp to the software inside the corresponding transmit descriptor,

thus connecting the timestamp automatically to the specific PTP frame. The 64-bit timestamp information is written to the TDES2 and TDES3 fields. T[†]

Receive Path Functions

The MAC captures the timestamp of all frames received on the PHY interface. The DMA returns the timestamp to the software in the corresponding receive descriptor. The timestamp is written only to the last receive descriptor.[†]

Timestamp Error Margin

According to the IEEE1588 specifications, a timestamp must be captured at the SFD of the transmitted and received frames at the PHY interface. Because the PHY interface receive and transmit clocks are not synchronous to the reference timestamp clock (clk_ptp_ref) a small amount of drift is introduced when a timestamp is moved between asynchronous clock domains. In the transmit path, the captured and reported timestamp has a maximum error margin of two PTP clocks, meaning that the captured timestamp has a reference timing source value that is occurred within two clocks after the SFD is transmitted on the PHY interface.

Similarly, in the receive path, the error margin is three PHY interface clocks, plus up to two PTP clocks. You can ignore the error margin due to the PHY interface clock by assuming that this constant delay is present in the system (or link) before the SFD data reaches the PHY interface of the MAC.[†]

Frequency Range of Reference Timing Clock

The timestamp information is transferred across asynchronous clock domains, from the EMAC clock domain to the FPGA clock domain. Therefore, a minimum delay is required between two consecutive timestamp captures. This delay is four PHY interface clock cycles and three PTP clock cycles. If the delay between two timestamp captures is less than this amount, the MAC does not take a timestamp snapshot for the second frame.

The maximum PTP clock frequency is limited by the maximum resolution of the reference time (20 ns resulting in 50 MHz) and the timing constraints achievable for logic operating on the PTP clock. In addition, the resolution, or granularity, of the reference time source determines the accuracy of the synchronization. Therefore, a higher PTP clock frequency gives better system performance.[†]

The minimum PTP clock frequency depends on the time required between two consecutive SFD bytes. Because the PHY interface clock frequency is fixed by the IEEE 1588 specification, the minimum PTP clock frequency required for proper operation depends on the operating mode and operating speed of the MAC.[†]

Table 17-18: Minimum PTP Clock Frequency Example

Mode	Minimum Gap Between Two SFDs	Minimum PTP Frequency
100-Mbps full-duplex operation	168 MII clocks (128 clocks for a 64-byte frame + 24 clocks of min IFG + 16 clocks of preamble)	$(3 * \text{PTP}) + (4 * \text{MII}) \leq 168 * \text{MII}$, that is, $\sim 0.5 \text{ MHz } (168 - 4) * 40 \text{ ns} \div 3 = 2180 \text{ ns period}$

Mode	Minimum Gap Between Two SFDs	Minimum PTP Frequency
1000-Mbps half duplex operation	24 GMII clocks (4 for a jam pattern sent just after SFD because of collision + 12 IFG + 8 preamble)	$(3 * \text{PTP}) + 4 * \text{GMII} \leq 24 * \text{GMII}$, that is, 18.75 MHz

Related Information**IEEE Standards Association**

For details about jam patterns, refer to the *IEEE Std 802.3 2008 Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, available on the IEEE Standards Association website.

IEEE 1588-2008 Advanced Timestamps

In addition to the basic timestamp features mentioned in IEEE 1588-2002 Timestamps, the EMAC supports the following advanced timestamp features defined in the IEEE 1588-2008 standard.[†]

- Supports the IEEE 1588-2008 (version 2) timestamp format.[†]
- Provides an option to take a timestamp of all frames or only PTP-type frames.[†]
- Provides an option to take a timestamp of event messages only.[†]
- Provides an option to take the timestamp based on the clock type: ordinary, boundary, end-to-end, or peer-to-peer.[†]
- Provides an option to configure the EMAC to be a master or slave for ordinary and boundary clock.[†]
- Identifies the PTP message type, version, and PTP payload in frames sent directly over Ethernet and sends the status.[†]
- Provides an option to measure sub-second time in digital or binary format.[†]

Related Information**IEEE Standards Association**

For more information about advanced timestamp features, refer to the *IEEE Standard 1588 - 2008 IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control System*, available on the IEEE Standards Association website.

Peer-to-Peer PTP Transparent Clock (P2P TC) Message Support

The IEEE 1588-2008 version supports Peer-to-Peer PTP (Pdelay) messages in addition to SYNC, Delay Request, Follow-up, and Delay Response messages.[†]

Clock Types

The EMAC supports the following clock types defined in the IEEE 1588-2008 standard:

- Ordinary clock[†]
- Boundary clock[†]
- End-to-End transparent clock[†]
- Peer-to-Peer transparent clock[†]

Ordinary Clock

The ordinary clock in a domain supports a single copy of the protocol. The ordinary clock has a single PTP state and a single physical port. In typical industrial automation applications, an ordinary clock is associated with an application device such as a sensor or an actuator. In telecom applications, the ordinary clock can be associated with a timing demarcation device. [†]

The ordinary clock can be a grandmaster or a slave clock. It supports the following features:[†]

- Sends and receives PTP messages. The timestamp snapshot can be controlled as described in the `Timestamp_Control` register.[†]
- Maintains the data sets such as timestamp values.[†]

The table below shows the messages for which you can take the timestamp snapshot on the receive side for Master and slave nodes. For an ordinary clock, you can take the snapshot of either of the following PTP message types: version 1 or version 2. You cannot take the snapshots for both PTP message types. You can take the snapshot by setting the control bit (`tsver2ena`) and selecting the snapshot mode in the `Timestamp_Control` register.[†]

Table 17-19: Ordinary Clock: PTP Messages for Snapshot[†]

Master	Slave
Delay_Req	SYNC

Boundary Clock

The boundary clock typically has several physical ports communicating with the network. The messages related to synchronization, master-slave hierarchy, and signaling terminate in the protocol engine of the boundary clock and are not forwarded. The PTP message type status given by the MAC helps you to identify the type of message and take appropriate action. The boundary clock is similar to the ordinary clock except for the following features:[†]

- The clock data sets are common to all ports of the boundary clock.[†]
- The local clock is common to all ports of the boundary clock. Therefore, the features of the ordinary clock are also applicable to the boundary clock.[†]

End-to-End Transparent Clock

The end-to-end transparent clock supports the end-to-end delay measurement mechanism between slave clocks and the master clock. The end-to-end transparent clock forwards all messages like normal bridge, router, or repeater. The residence time of a PTP packet is the time taken by the PTP packet from the ingress port to the egress port.[†]

The residence time of a SYNC packet inside the end-to-end transparent clock is updated in the correction field of the associated Follow_Up PTP packet before it is transmitted. Similarly, the residence time of a Delay_Req packet inside the end-to-end transparent clock is updated in the correction field of the associated Delay_Resp PTP packet before it is transmitted. Therefore, the snapshot needs to be taken at both ingress and egress ports only for PTP messages SYNC or Delay_req. You can take the snapshot by setting the snapshot select bits (`snaptypesel`) to b'10 in the `Timestamp_Control` register.[†]

The `snaptypesel` bits, along with bits 15 and 14 in the `Timestamp_Control` register, decide the set of PTP packet types for which a snapshot needs to be taken. The encoding is shown in the table below:[†]

Table 17-20: Timestamp Snapshot Dependency on Register Bits[†]

X is defined as a "don't care" in the table.

snaptypsel (bits[17:16])	tsmstrena (bit 15)	tsevtena (bit 14)	PTP Messages
0x0	X	0	SYNC, Follow_Up, Delay_Req, Delay_Resp
0x0	0	1	SYNC
0x0	1	1	Delay_Req
0x1	X	0	SYNC, Follow_Up, Delay_Req, Delay_Resp, Pdelay_Req, Pdelay_Resp, Pdelay_Resp_Follow_Up
0x1	0	1	SYNC, Pdelay_Req, Pdelay_Resp
0x1	1	1	Delay_Req, Pdelay_Req, Pdelay_Resp
0x2	X	X	SYNC, Delay_Req
0x3	X	X	Pdelay_Req, Pdelay_Resp

Peer-to-Peer Transparent Clock

The peer-to-peer transparent clock differs from the end-to-end transparent clock in the way it corrects and handles the PTP timing messages. In all other aspects, it is identical to the end-to-end transparent clock.[†]

In the peer-to-peer transparent clock, the computation of the link delay is based on an exchange of Pdelay_Req, Pdelay_Resp, and Pdelay_Resp_Follow_Up messages with the link peer. The residence time of the Pdelay_Req and the associated Pdelay_resp packets is added and inserted into the correction field of the associated Pdelay_Resp_Followup packet.[†]

Therefore, support for taking snapshot for the event messages related to Pdelay is added as shown in the table below.[†]

Table 17-21: Peer-to-Peer Transparent Clock: PTP Messages for Snapshot[†]

PTP Messages
SYNC
Pdelay_Req
Pdelay_Resp

You can take the snapshot by setting the snapshot select bits (snaptypsel) to b'11 in the Timestamp Control register.[†]

Reference Timing Source

The EMAC supports the following reference timing source features defined in the IEEE 1588-2008 standard:

- 48-bit seconds field[†]
- Fixed pulse-per-second output[†]
- Flexible pulse-per-second output[†]
- Auxiliary snapshots (timestamps) with external events

Transmit Path Functions

The advanced timestamp feature is supported through the descriptors format.

Receive Path Functions

The MAC processes the received frames to identify valid PTP frames. You can control the snapshot of the time to be sent to the application, by using the following options:[†]

- Enable timestamp for all frames.[†]
- Enable timestamp for IEEE 1588 version 2 or version 1 timestamp.[†]
- Enable timestamp for PTP frames transmitted directly over Ethernet or UDP/IP Ethernet.[†]
- Enable timestamp snapshot for the received frame for IPv4 or IPv6.[†]
- Enable timestamp snapshot for EVENT messages (SYNC, DELAY_REQ, PDELAY_REQ, or PDELAY_RESP) only.[†]
- Enable the node to be a master or slave and select the timestamp type to control the type of messages for which timestamps are taken.[†]

The DMA returns the timestamp to the software inside the corresponding transmit or receive descriptor.

Auxiliary Snapshot

The auxiliary snapshot feature allows you to store a snapshot (timestamp) of the system time based on an external event. The event is considered to be the rising edge of the sideband signal `ptp_aux_ts_trig_i` from the FPGA. One auxiliary snapshot input is available. The depth of the auxiliary snapshot FIFO buffer is 16.

The timestamps taken for any input are stored in a common FIFO buffer. The host can read Register 458 (Timestamp Status Register) to know which input's timestamp is available for reading at the top of this FIFO buffer.

Only 64-bits of the timestamp are stored in the FIFO. You can read the upper 16-bits of seconds from Register 457 (System Time - Higher Word Seconds Register) when it is present. When a snapshot is stored, the MAC indicates this to the host with an interrupt. The value of the snapshot is read through a FIFO register access. If the FIFO becomes full and an external trigger to take the snapshot is asserted, then a snapshot trigger-missed status (ATSSTM) is set in Register 458 (Timestamp Status Register). This indicates that the latest auxiliary snapshot of the timestamp was not stored in the FIFO. The latest snapshot is not written to the FIFO when it is full. When a host reads the 64-bit timestamp from the FIFO, the space becomes available to store the next snapshot. You can clear a FIFO by setting Bit 19 (ATSFC) in Register 448 (Timestamp Control Register). When multiple snapshots are present in the FIFO, the count is indicated in Bits [27:25], ATSNS, of Register 458 (Timestamp Status Register).[†]

IEEE 802.3az Energy Efficient Ethernet

Energy Efficient Ethernet (EEE) standardized by IEEE 802.3-az, version D2.0 is supported by the EMAC. It is supported by the MAC operating in 10/100/1000 Mbps rates. EEE is only supported when the EMAC

is configured to operate with the RGMII PHY interface operating in full-duplex mode. It cannot be used in half-duplex mode.

EEE enables the MAC to operate in Low-Power Idle (LPI) mode. Either end point of an Ethernet link can disable functionality to save power during periods of low link utilization. The MAC controls whether the system should enter or exit LPI mode and communicates this information to the PHY.[†]

Related Information

IEEE 802.3 Ethernet Working Group

For details about the *IEEE 802.3az Energy Efficient Ethernet standard*, refer to the IEEE 802.3 Ethernet Working Group website.[†]

LPI Timers

Two timers internal to the EMAC are associated with LPI mode:

- LPI Link Status (LS) Timer[†]
- LPI Time Wait (TW) Timer[†]

The LPI LS timer counts, in ms, the time expired since the link status has come up. This timer is cleared every time the link goes down and is incremented when the link is up again and the terminal count as programmed by the software is reached. The PHY interface does not assert the LPI pattern unless the terminal count is reached. This protocol ensures a minimum time for which no LPI pattern is asserted after a link is established with the remote station. This period is defined as one second in the IEEE standard 802.3-az, version D2.0. The LPI LS timer is 10 bits wide, so the software can program up to 1023 ms.[†]

The LPI TW timer counts, in μ s, the time expired since the deassertion of LPI. The terminal count of the timer is the value of resolved transmit TW that is the auto-negotiated time after which the MAC can resume the normal transmit operation. The LPI TW timer is 16 bits wide, so the software can program up to 65535 μ s.[†]

The EMAC generates the LPI interrupt when the transmit or receive channel enters or exits the LPI state.[†]

Checksum Offload

Communication protocols such as TCP and UDP implement checksum fields, which help determine the integrity of data transmitted over a network. Because the most widespread use of Ethernet is to encapsulate TCP and UDP over IP datagrams, the EMAC has a Checksum Offload Engine (COE) to support checksum calculation and insertion in the transmit path, and error detection in the receive path. Supported offloading types:

- Transmit IP header checksum[†]
- Transmit TCP/UDP/ICMP checksum[†]
- Receive IP header checksum[†]
- Receive full checksum[†]

Frame Filtering

The EMAC implements the following types of filtering for receive frames.

Source Address or Destination Address Filtering

The Address Filtering Module checks the destination and source address field of each incoming packet.[†]

Unicast Destination Address Filter

Up to 128 MAC addresses for unicast perfect filtering are supported. The filter compares all 48 bits of the received unicast address with the programmed MAC address for any match. Default MacAddr0 is always enabled, other addresses MacAddr1–MacAddr127 are selected with an individual enable bit. For MacAddr1–MacAddr31 addresses, you can mask each byte during comparison with the corresponding received DA byte. This enables group address filtering for the DA. The MacAddr32–MacAddr127 addresses do not have mask control and all six bytes of the MAC address are compared with the received six bytes of DA.[†]

In hash filtering mode, the filter performs imperfect filtering for unicast addresses using a 256-bit hash table. It uses the upper ten bits of the CRC of the received destination address to index the content of the hash table. A value of 0 selects Bit 0 of the selected register, and a value of 111111 binary selects Bit 63 of the Hash Table register. If the corresponding bit is set to one, the unicast frame is said to have passed the hash filter; otherwise, the frame has failed the hash filter.[†]

Multicast Destination Address Filter

The MAC can be programmed to pass all multicast frames. In Perfect Filtering mode, the multicast address is compared with the programmed MAC Destination Address registers (1–31). Group address filtering is also supported. In hash filtering mode, the filter performs imperfect filtering using a 256-bit hash table. For hash filtering, it uses the upper ten bits of the CRC of the received multicast address to index the contents of the hash table. A value of 0 selects Bit 0 of the selected register and a value of 111111 binary selects Bit 63 of the Hash Table register. If the corresponding bit is set to one, then the multicast frame is said to have passed the hash filter; otherwise, the frame has failed the hash filter.[†]

Hash or Perfect Address Filter

The filter can be configured to pass a frame when its DA matches either the hash filter or the Perfect filter. This configuration applies to both unicast and multicast frames.[†]

Broadcast Address Filter

The filter does not filter any broadcast frames in the default mode. However, if the MAC is programmed to reject all broadcast frames, the filter drops any broadcast frame.[†]

Unicast Source Address Filter

The MAC can also perform a perfect filtering based on the source address field of the received frames. Group filtering with SA is also supported. You can filter a group of addresses by masking one or more bytes of the address.[†]

Inverse Filtering Operation (Invert the Filter Match Result at Final Output)

For both Destination and Source address filtering, there is an option to invert the filter-match result at the final output. The result of the unicast or multicast destination address filter is inverted in this mode.[†]

Destination and Source Address Filtering Summary

The tables below summarize the destination and source address filtering based on the type of frames received and the configuration of bits within the Mac_Frame_Filter register.[†]

Table 17-22: Destination Address Filtering[†]

Note: The "X" in the table represents a "don't care" term.

Frame Type	PR	HPF	HUC	DAIF	HMC	PM	DBF	Destination Address Filter Operation
Broadcast	1	X	X	X	X	X	X	Pass
	0	X	X	X	X	X	0	Pass
	0	X	X	X	X	X	1	Fail
Unicast	1	X	X	X	X	X	X	Pass all frames
	0	X	0	0	X	X	X	Pass on Perfect/Group filter match
	0	X	0	1	X	X	X	Fail on Perfect/Group filter match
	0	0	1	0	X	X	X	Pass on Hash filter match
	0	0	1	1	X	X	X	Fail on Hash filter match
	0	1	1	0	X	X	X	Pass on Hash or Perfect/Group filter match
	0	1	1	1	X	X	X	Fail on Hash or Perfect/Group filter match
Multicast	1	X	X	X	X	X	X	Pass all frames
	X	X	X	X	X	1	X	Pass all frames
	0	X	X	0	0	0	X	Pass on Perfect/Group filter match and drop Pause frames if PCF= 0X
	0	0	X	0	1	0	X	Pass on Hash filter match and drop Pause frames if PCF = 0X
	0	1	X	0	1	0	X	Pass on Hash or Perfect/Group filter match and drop Pause frames if PCF = 0X
	0	X	X	1	0	0	X	Fail on Perfect/Group filter match and drop Pause frames if PCF=0X
	0	0	X	1	1	0	X	Fail on Hash filter match and drop Paus frames if PCF = 0X
	0	1	X	1	1	0	X	Fail Hash on Perfect/Group filter match and drop Pause frames if PCF = 0X

Table 17-23: Source Address Filtering[†]

Frame Type	PR	SAIF	SBF	Source Address Filter Operation
Unicast	1	X	X	Pass all frames
	0	0	0	Pass status on Perfect or Group filter match but do not drop frames that fail.
	0	1	0	Fail on Perfect or Group filter match but do not drop frame
	0	0	1	Pass on Perfect or Group filter match and drop frames that fail
	0	1	1	Fail on Perfect or Group filter match and drop frames that fail

VLAN Filtering

The EMAC supports the two kinds of VLAN filtering:

- VLAN tag-based filtering[†]
- VLAN hash filtering[†]

VLAN Tag-Based Filtering

In the VLAN tag-based frame filtering, the MAC compares the VLAN tag of the received frame and provides the VLAN frame status to the application. Based on the programmed mode, the MAC compares the lower 12 bits or all 16 bits of the received VLAN tag to determine the perfect match. If VLAN tag filtering is enabled, the MAC forwards the VLAN-tagged frames along with VLAN tag match status and drops the VLAN frames that do not match. You can also enable the inverse matching for VLAN frames. In addition, you can enable matching of SVLAN tagged frames along with the default Customer Virtual Local Area Network (C-VLAN) tagged frames.[†]

VLAN Hash Filtering with a 16-Bit Hash Table

The MAC provides VLAN hash filtering with a 16-bit hash table. The MAC also supports the inverse matching of the VLAN frames. In inverse matching mode, when the VLAN tag of a frame matches the perfect or hash filter, the packet should be dropped. If the VLAN perfect and VLAN hash match are enabled, a frame is considered as matched if either the VLAN hash or the VLAN perfect filter matches. When inverse match is set, a packet is forwarded only when both perfect and hash filters indicate mismatch.[†]

Layer 3 and Layer 4 Filters

Layer 3 filtering refers to source address and destination address filtering. Layer 4 filtering refers to source port and destination port filtering. The frames are filtered in the following ways:[†]

- Matched frames[†]
- Unmatched frames[†]
- Non-TCP or UDP IP frames[†]

Matched Frames

The MAC forwards the frames, which match all enabled fields, to the application along with the status. The MAC gives the matched field status only if one of the following conditions is true:[†]

- All enabled Layer 3 and Layer 4 fields match.[†]
- At least one of the enabled field matches and other fields are bypassed or disabled.[†]

Using the CSR set, you can define up to four filters, identified as filter 0 through filter 3. When multiple Layer 3 and Layer 4 filters are enabled, any filter match is considered as a match. If more than one filter matches, the MAC provides status of the lowest filter with filter 0 being the lowest and filter 3 being the highest. For example, if filter 0 and filter 1 match, the MAC gives the status corresponding to filter 0.[†]

Unmatched Frames

The MAC drops the frames that do not match any of the enabled fields. You can use the inverse match feature to block or drop a frame with specific TCP or UDP over IP fields and forward all other frames. You can configure the EMAC so that when a frame is dropped, it receives a partial frame with appropriate abort status or drops it completely.[†]

NonTCP or UDP IP Frames

By default, all non-TCP or UDP IP frames are bypassed from the Layer 3 and Layer 4 filters. You can optionally program the MAC to drop all non-TCP or UDP over IP frames.[†]

Layer 3 and Layer 4 Filters Register Set

The MAC implements a set of registers for Layer 3 and Layer 4 based frame filtering. In this register set, there is a control register for frame filtering and five address registers.[†]

You can configure the MAC to have up to four such independent set of registers.[†]

The registers available for programming are as follows:

- L3_L4_Control0 through L3_L4_Control3 registers: Layer and Layer 4 Control registers
- Layer4_Address0 through Layer4_Address3 registers: Layer 4 Address registers
- Layer3_Address0_Reg0 through Layer3_Address0_Reg3 registers: Layer 3 Address 0 registers
- Layer3_Address1_Reg0 through Layer3_Address1_Reg3 registers: Layer 3 Address 1 registers
- Layer3_Address2_Reg0 through Layer3_Address2_Reg3 registers: Layer 3 Address 2 registers
- Layer3_Address3_Reg0 through Layer3_Address3_Reg3 registers: Layer 3 Address 3 registers

Related Information

[Ethernet MAC Address Map and Register Definitions](#) on page 17-72

Layer 3 Filtering

The EMAC supports perfect matching or inverse matching for the IP Source Address and Destination Address. In addition, you can match the complete IP address or mask the lower bits.[†]

For IPv6 frames filtering, you can enable the last four data registers of a register set to contain the 128-bit IP Source Address or IP Destination Address. The IP Source or Destination Address should be programmed in the order defined in the IPv6 specification. The specification requires that you program the first byte of the received frame IP Source or Destination Address in the higher byte of the register. Subsequent registers should follow the same order.[†]

For IPv4 frames filtering, you can enable the second and third data registers of a register set to contain the 32-bit IP Source Address and IP Destination Address. The remaining two data registers are reserved. The IP Source and Destination Address should be programmed in the order defined in the IPv4 specification. The specification requires that you program the first byte of received frame IP Source and Destination Address in the higher byte of the respective register.[†]

Layer 4 Filtering

The EMAC supports perfect matching or inverse matching for TCP or UDP Source and Destination Port numbers. However, you can program only one type (TCP or UDP) at a time. The first data register contains the 16-bit Source and Destination Port numbers of TCP or UDP, that is, the lower 16 bits for Source Port number and higher 16 bits for Destination Port number. [†]

The TCP or UDP Source and Destination Port numbers should be programmed in the order defined in the TCP or UDP specification, that is, the first byte of TCP or UDP Source and Destination Port number in the received frame is in the higher byte of the register. [†]

Clocks and Resets

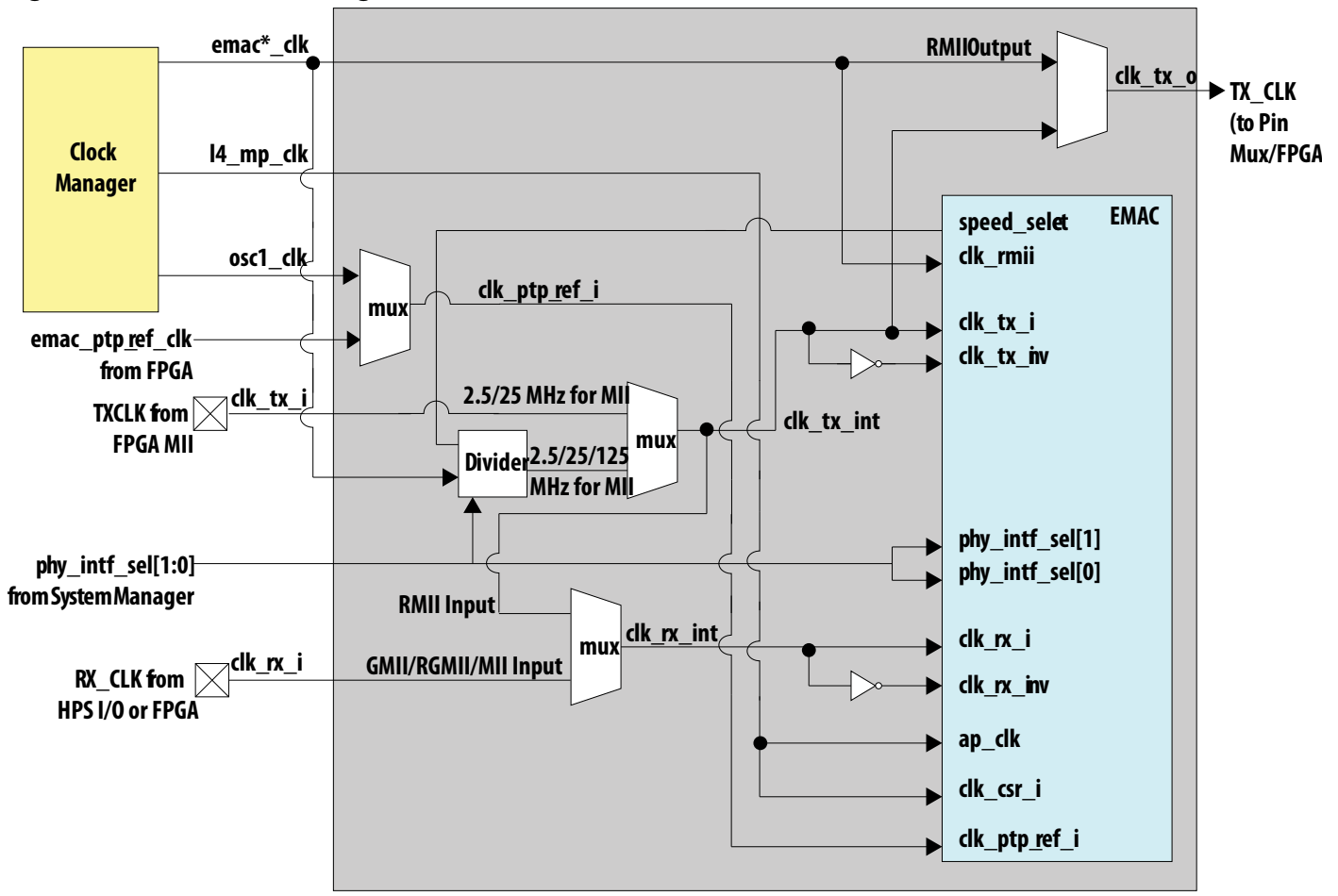
Clock Structure

The Ethernet Controller received five input clocks and generates one output clock.

The EMAC controller provides the clock division and muxing necessary to generate the proper clocks for each of the PHY modes and data rates. The clock domains to the Ethernet controller are as follows

- l4_mp_clk clock
- EMAC RX clock
- EMAC TX clock
- clk_ptp_ref

Figure 17-15: EMAC Clock Diagram



The diagram below summarizes the clock domains of the EMAC module:

Clock Gating for EEE

For the RGMII PHY interface, you can gate the transmit clock for Energy Efficient Ethernet (EEE) applications.

Related Information

[Programming Guidelines for Energy Efficient Ethernet](#) on page 17-69

Reset

The EMAC module accepts a single reset input, `emac_rst_n`, which is active low.

Note: In all modes, the EMAC core depends on the PHY clocks to be active for the internal EMAC clock sources to be valid.

Taking the Ethernet MAC Out of Reset

When a cold or warm reset is issued in the HPS, the Reset Manager resets the EMAC module and holds it in reset until software releases it.

After the MPU boots up, it can deassert the reset signal by clearing the appropriate bits in the Reset Manager's corresponding reset register. Before deasserting the reset signal, you must make sure the PHY

interface type and all other corresponding EMAC settings in the System Manager have been configured. For details about reset registers, refer to the "Module Reset Signals" section in the *Reset Manager* chapter. For more information about EMAC configuration in the System Manager, refer to the "System Level EMAC Configuration Registers" section.

Related Information

[Module Reset Signals](#) on page 4-5

For more information about reset registers refer to this section in the *Reset Manager* chapter.

Interrupts

Interrupts are generated as a result of specific events in the EMAC and external PHY device. The interrupt status register indicates all conditions which may trigger an interrupt and the interrupt enable register determines which interrupts can propagate.

Ethernet MAC Programming Model

The initialization and configuration of the EMAC and its interface is a multi-step process that includes system register programming in the System Manager and Clock Manager and configuration of clocks in multiple domains.

Note: When the EMAC interfaces to HPS I/O and register content is being transferred to a different clock domain after a write operation, no further writes should occur to the same location until the first write is updated. Otherwise, the second write operation does not get updated to the destination clock domain. Thus, the delay between two writes to the same register location should be at least 4 cycles of the destination clock (PHY receive clock, PHY transmit clock, or PTP clock).[†] If the CSR is accessed multiple times quickly, you must ensure that a minimum number of destination clock cycles have occurred between accesses.

Note: If the EMAC signals are routed through the FPGA fabric and it is assumed that the transmit clock supplied by the FPGA fabric switches within 6 transmit clock cycles, then the minimum time required between two write accesses to the same register is 10 transmit clock cycles.

System Level EMAC Configuration Registers

In addition to the registers in the Ethernet Controller, there are other system level registers in the Clock Manager, System Manager and Reset Manager that must be programmed in order to configure the EMAC and its interfaces.

The following table gives a summary of the important System Manager clock register bits that control operation of the EMAC. These register bits are static signals that must be set while the corresponding EMAC is in reset.

Table 17-24: System Manager Clock and Interface Settings

Register.Field	Description
ctrl.ptpclkssel_0 ctrl.ptpclkssel_1	1588 PTP reference clock. This bit selects the source of the 1588 PTP reference clock. 0x0= osc1_clk (default from Clock Manager) 0x1= fpga_ptp_ref_clk (from FPGA fabric; in this case, the FPGA must be in usermode with an active reference clock)
ctrl.physel_0 ctrl.physel_1	PHY Interface Select. These two bits set the PHY mode. 0x0= GMII or MII 0x1= RGMII 0x2= RMII (default) Note: Selecting the 0x0 encoding routes the GMII/MII signals to the FPGA fabric only and selecting the 0x1 encoding routes the RGMII signals to the HPS only. 0x2 is not a valid encoding and depending on the interface selected, must be changed to 0x0 or 0x1 out of reset.

The following table summarizes the important System Manager configuration register bits. All of the fields, except the AXI cache settings, are assumed to be static and must be set before the EMAC is brought out of reset. If the FPGA interface is used, the FPGA must be in user mode and enabled with the appropriate clock signals active before the EMAC can be brought out of reset.

Table 17-25: System Manager Static Control Settings

Register.Field	Description
module.emac_0 module.emac_1	FPGA interface to EMAC disable. This field is used to disable signals from the FPGA to the EMAC modules that could potentially interfere with their normal operation 0x0= Disable (default) 0x1=Enable
l3master.awcache_1 l3master.awcache_0 l3master.arcache_1 l3master.arcache_0	EMAC AXI Master AxCACHE settings. It is recommended that these bits are set while the EMAC is idle or in reset.

Various registers within the Clock Manager must also be configured in order for the EMAC controller to perform properly.

Table 17-26: Clock Manager Settings

Register.Field	Description
emac0clk.cnt emac1clk.cnt	EMAC clock control. The cnt value in this register is used to divide the PLL VCO frequency to generate emac0clk and emac1clk.
en.emac0clk en.emac1clk	emac0clk and emac1clk output enable.

EMAC FPGA Interface Initialization

To initialize the Ethernet controller to use the FPGA GMII/MII interface, specific software steps must be followed.

In general, the FPGA interface must be active in user mode with valid PHY clocks, the Ethernet Controller must be in a reset state during static configuration and the clock must be active and valid before the Ethernet Controller is brought out of reset.

1. After the HPS is released from cold or warm reset, reset the Ethernet Controller module by setting the appropriate emac bit in the permodrst register in the Reset Manager.
2. Configure the EMAC Controller clock to 250 MHz by programming the appropriate cnt value in the emac*clk register in the Clock Manager.
3. Bring the Ethernet PHY out of reset to allow PHY to generate RX clocks and TX clocks.

When using FPGA GMII/MII interface, you must have a stable RX clock (emac_clk_rx_i) and TX clock (emac_clk_tx_i) supply from PHY to EMAC before bringing EMAC out of reset.

There are no registers to verify, but you can create the following custom logic block to cross check:

- You can use Signal Tap to check, or create a simple counter block with the RX clock and TX clock as clock source to check if it runs.
4. If the PTP clock source is from the FPGA, ensure that the FPGA f2s_ptp_ref_clk is active.
 5. The soft GMII/MII adaptor must be loaded with active clocks propagating. The FPGA must be configured to user mode and a reset to the user soft FPGA IP may be required to propagate the PHY clocks to the HPS.
 6. Once all clock sources are valid, apply the following clock settings:
 - a. Program the physe*_field in the ctrl register of the System Manager (EMAC Group) to 0x0 to select the GMII/MII PHY interface.
 - b. If the PTP clock source is from the FPGA, set the ptpclkse*_bit in the ctrl register (EMAC group) of the System Manager to be 0x1.
 - c. Enable the Ethernet Controller FPGA interface by setting the emac_* bit in the module register of the System Manager (FPGA Interface group).
 7. Configure all of the EMAC static settings if the user requires a different setting from the default value. These settings include AXI AxCACHE signal values which are programmed in l3 register in the EMAC group of the System Manager.
 8. Execute a register read back to confirm the clock and static configuration settings are valid.
 9. After confirming the settings are valid, software can clear the emac bit in the permodrst register of the Reset Manager to bring the EMAC out of reset.

When these steps are completed, general Ethernet controller and DMA software initialization and configuration can continue.

Note: These same steps can be applied to convert the HPS GMII to an RGMII, RMII or SGMII interface through the FPGA, except that in step 5 during FPGA configuration, you would load the appropriate soft adaptor for the interface and apply reset to it as well. The PHY interface select encoding would remain as 0x0. For the SGMII interface additional external transceiver logic would be required. Routing the Ethernet signals through the FPGA is useful for designs that are pin-limited in the HPS.

EMAC HPS Interface Initialization

To initialize the Ethernet controller to use the HPS interface, specific software steps must be followed including selecting the correct PHY interface through the System Manager.

In general, the Ethernet Controller must be in a reset state during static configuration and the clock must be active and valid before the Ethernet Controller is brought out of reset.

1. After the HPS is released from cold or warm reset, reset the Ethernet Controller module by setting the appropriate emac bit in the permodrst register in the Reset Manager.
2. Configure the EMAC Controller clock to 250 MHz by programming the appropriate cnt value in the emac*clk register in the Clock Manager.
3. Bring the Ethernet PHY out of reset to allow PHY to generate RX clocks.

There are no registers to verify, but you can create the following custom logic block to cross check:

- If the RX clock is routed through FPGA IO—you can use Signal Tap to check, or create a simple counter block with the RX clock as clock source to check if it runs.
 - If the RX clock is routed as HPS IO—you need to explore if the kernel application code is able to source through RX clock to check its status.
4. When all the clocks are valid, program the following clock settings:
 - a. Set the physe1_* field in the ctrl register of the System Manager (EMAC Group) to 0x1 to select the RGMII PHY interface.
 - b. Disable the Ethernet Controller FPGA interfaces by clearing the emac_* bit in the module register of the System Manager (FPGA Interface group).
 5. Configure all of the EMAC static settings if the user requires a different setting from the default value. These settings include AXI AxCACHE signal values, which are programmed in l3 register in the EMAC group of the System Manager.
 6. Execute a register read back to confirm the clock and static configuration settings are valid.
 7. After confirming the settings are valid, software can clear the emac bit in the permodrst register of the Reset Manager to bring the EMAC out of reset..

When these steps are completed, general Ethernet controller and DMA software initialization and configuration can continue.

DMA Initialization

This section provides the instructions for initializing the DMA registers in the proper sequence. This initialization sequence can be done after the EMAC interface initialization has been completed. Perform the following steps to initialize the DMA:

1. Provide a software reset to reset all of the EMAC internal registers and logic. (DMA Register 0 (Bus Mode Register) – bit 0).[†]
2. Wait for the completion of the reset process (poll bit 0 of the DMA Register 0 (Bus Mode Register), which is only cleared after the reset operation is completed).[†]
3. Poll the bits of Register 11 (AXI Status) to confirm that all previously initiated (before software reset) or ongoing transactions are complete.

Note: If the application cannot poll the register after soft reset (because of performance reasons), then it is recommended that you continue with the next steps and check this register again (as mentioned in step 12 on page 1-67) before triggering the DMA operations.[†]

4. Program the following fields to initialize the Bus Mode Register by setting values in DMA Register 0 (Bus Mode Register):[†]
 - Mixed Burst and AAL
 - Fixed burst or undefined burst[†]
 - Burst length values and burst mode values[†]
 - Descriptor Length (only valid if Ring Mode is used)[†]
5. Program the interface options in Register 10 (AXI Bus Mode Register). If fixed burst-length is enabled, then select the maximum burst-length possible on the bus (bits[7:1]).^{†(54)}
6. Create a proper descriptor chain for transmit and receive. In addition, ensure that the receive descriptors are owned by DMA (bit 31 of descriptor should be set). When OSF mode is used, at least two descriptors are required.
7. Make sure that your software creates three or more different transmit or receive descriptors in the chain before reusing any of the descriptors.[†]
8. Initialize receive and transmit descriptor list address with the base address of the transmit and receive descriptor (Register 3 (Receive Descriptor List Address Register) and Register 4 (Transmit Descriptor List Address Register) respectively).[†]
9. Program the following fields to initialize the mode of operation in Register 6 (Operation Mode Register):
 - Receive and Transmit Store And Forward[†]
 - Receive and Transmit Threshold Control (RTC and TTC)[†]
 - Hardware Flow Control enable[†]
 - Flow Control Activation and De-activation thresholds for MTL Receive and Transmit FIFO buffers (RFA and RFD)[†]
 - Error frame and undersized good frame forwarding enable[†]
 - OSF Mode[†]
10. Clear the interrupt requests, by writing to those bits of the status register (interrupt bits only) that are set. For example, by writing 1 into bit 16, the normal interrupt summary clears this bit (DMA Register 5 (Status Register)).[†]
11. Enable the interrupts by programming Register 7 (Interrupt Enable Register).[†]

Note: Perform step 12 on page 1-67 only if you did not perform step 3 on page 1-67.[†]
12. Read Register 11 (AHB or AXI Status) to confirm that all previous transactions are complete.[†]

⁽⁵⁴⁾ The Arria V implementation supports bits [3:1]

Note: If any previous transaction is still in progress when you read the Register 11 (AXI Status), then it is strongly recommended to check the slave components addressed by the master interface.[†]

13. Start the receive and transmit DMA by setting SR (bit 1) and ST (bit 13) of the control register (DMA Register 6 (Operation Mode Register)).[†]

EMAC Initialization and Configuration

The following EMAC configuration operations can be performed after DMA initialization. If the EMAC initialization and configuration is done before the DMA is set up, then enable the MAC receiver (last step below) only after the DMA is active. Otherwise, the received frame could fill the RX FIFO buffer and overflow.

1. Program the GMII Address Register (offset 0x10) for controlling the management cycles for the external PHY. Bits[15:11] of the GMII Address Register are written with the Physical Layer Address of the PHY before reading or writing. Bit 0 indicates if the PHY is busy and is set before reading or writing to the PHY management interface.[†]
2. Read the 16-bit data of the GMII Data Register from the PHY for link up, speed of operation, and mode of operation, by specifying the appropriate address value in bits[15:11] of the GMII Address Register.[†]
3. Provide the MAC address registers (MAC Address0 High Register through MAC Address15 High Register and MAC Address0 Low Register through MAC Address15 Low Register).
4. Program the Hash Table Registers 0 through 7 (offset 0x500 to 0x51C).
5. Program the following fields to set the appropriate filters for the incoming frames in the MAC Frame Filter Register:[†]
 - Receive All[†]
 - Promiscuous mode[†]
 - Hash or Perfect Filter[†]
 - Unicast, multicast, broadcast, and control frames filter settings[†]
6. Program the following fields for proper flow control in the Flow Control Register:[†]
 - Pause time and other pause frame control bits[†]
 - Receive and Transmit Flow control bits[†]
 - Flow Control Busy/Backpressure Activate[†]
7. Program the Interrupt Mask Register bits, as required and if applicable for your configuration.[†]
8. Program the appropriate fields in MAC Configuration Register to configure receive and transmit operation modes. After basic configuration is written, set bit 3 (TE) and bit 2 (RE) in this register to enable the receive and transmit state machines.[†]

Note: Do not change the configuration (such as duplex mode, speed, port, or loopback) when the EMAC DMA is actively transmitting or receiving. Software should change these parameters only when the EMAC DMA transmitter and receiver are not active.

Performing Normal Receive and Transmit Operation

For normal operation, perform the following steps:[†]

1. For normal transmit and receive interrupts, read the interrupt status. Then, poll the descriptors, reading the status of the descriptor owned by the Host (either transmit or receive). †
2. Set appropriate values for the descriptors, ensuring that transmit and receive descriptors are owned by the DMA to resume the transmission and reception of data. †
3. If the descriptors are not owned by the DMA (or no descriptor is available), the DMA goes into SUSPEND state. The transmission or reception can be resumed by freeing the descriptors and issuing a poll demand by writing 0 into the TX/RX poll demand registers, (Register 1 (Transmit Poll Demand Register) and Register 2 (Receive Poll Demand Register)). †
4. The values of the current host transmitter or receiver descriptor address pointer can be read for the debug process (Register 18 (Current Host Transmit Descriptor Register) and Register 19 (Current Host Receive Descriptor Register)). †
5. The values of the current host transmit buffer address pointer and receive buffer address pointer can be read for the debug process (Register 20 (Current Host Transmit Buffer Address Register) and Register 21 (Current Host Receive Buffer Address Register)). †

Stopping and Starting Transmission

Perform the following steps to pause the transmission for some time: †

1. Disable the transmit DMA (if applicable), by clearing bit 13 (Start or Stop Transmission Command) of Register 6 (Operation Mode Register). †
2. Wait for any previous frame transmissions to complete. You can check this by reading the appropriate bits of Register 9 (Debug Register). †
3. Disable the EMAC transmitter and EMAC receiver by clearing Bit 3 (TE) and Bit 2 (RE) in Register 0 (MAC Configuration Register). †
4. Disable the receive DMA (if applicable), after making sure that the data in the RX FIFO buffer is transferred to the system memory (by reading Register 9 (Debug Register). †
5. Make sure that both the TX FIFO buffer and RX FIFO buffer are empty. †
6. To re-start the operation, first start the DMA and then enable the EMAC transmitter and receiver. †

Programming Guidelines for Energy Efficient Ethernet

Entering and Exiting the TX LPI Mode

The Energy Efficient Ethernet (EEE) feature is available in the EMAC. To use it, perform the following steps during EMAC initialization:

1. Read the PHY register through the MDIO interface, check if the remote end has the EEE capability, and then negotiate the timer values. †
2. Program the PHY registers through the MDIO interface (including the RX_CLK_stoppable bit that indicates to the PHY whether to stop the RX clock in LPI mode.) †
3. Program Bits[16:5], LST, and Bits[15:0], TWT, in Register 13 (LPI Timers Control Register). †
4. Read the link status of the PHY chip by using the MDIO interface and update Bit 17 (PLS) of Register 12 (LPI Control and Status Register) accordingly. This update should be done whenever the link status in the PHY changes. †
5. Set Bit 16 (LPIEN) of Register 12 (LPI Control and Status Register) to make the MAC enter the LPI state. The MAC enters the LPI mode after completing the transmission in progress and sets Bit 0 (TLPIEN). †

Note: To make the MAC enter the LPI state only after it completes the transmission of all queued frames in the TX FIFO buffer, you should set Bit 19 (LPITXA) in Register 12 (LPI Control and Status Register). †

Note: To switch off the transmit clock during the LPI state, use the `sbd_tx_clk_gating_ctrl_o` signal for gating the clock input. †

Note: To switch off the CSR clock or power to the rest of the system during the LPI state, you should wait for the TLPIEN interrupt of Register 12 (LPI Control and Status Register) to be generated. Restore the clocks before performing step 6 on page 1-70 when you want to come out of the LPI state. †

6. Clear Bit 16 (LPIEN) of Register 12 (LPI Control and Status Register) to bring the MAC out of the LPI state. †

The MAC waits for the time programmed in Bits [15:0], TWT, before setting the TLPIEX interrupt status bit and resuming the transmission. †

Gating Off the CSR Clock in the LPI Mode

You can gate off the CSR clock to save the power when the MAC is in the Low-Power Idle (LPI) mode. †

Gating Off the CSR Clock in the RX LPI Mode

The following operations are performed when the MAC receives the LPI pattern from the PHY. †

1. The MAC RX enters the LPI mode and the RX LPI entry interrupt status [RLPIEN interrupt of Register 12 (LPI_Control_Status)] is set. †
2. The interrupt pin (`sbd_intr_o`) is asserted. The `sbd_intr_o` interrupt is cleared when the host reads the Register 12 (LPI_Control_Status). †

After the `sbd_intr_o` interrupt is asserted and the MAC TX is also in the LPI mode, you can gate off the CSR clock. If the MAC TX is not in the LPI mode when you gate off the CSR clock, the events on the MAC transmitter do not get reported or updated in the CSR. †

For restoring the CSR clock, wait for the LPI exit indication from the PHY after which the MAC asserts the LPI exit interrupt on `lpi_intr_o` (synchronous to `clk_rx_i`). The `lpi_intr_o` interrupt is cleared when Register 12 is read. †

Gating Off the CSR Clock in the TX LPI Mode

The following operations are performed when Bit 16 (LPIEN) of Register 12 (LPI Control and Status Register) is set: †

1. The Transmit LPI Entry interrupt (TLPIEN bit of Register 12) is set. †
2. The interrupt pin (`sbd_intr_o`) is asserted. The `sbd_intr_o` interrupt is cleared when the host reads the Register 12. †

After the `sbd_intr_o` interrupt is asserted and the MAC RX is also in the LPI mode, you can gate off the CSR clock. If the MAC RX is not in the LPI mode when you gate off the CSR clock, the events on the MAC receiver do not get reported or updated in the CSR. †

To restore the CSR clock, switch on the CSR clock when the MAC has to come out of the TX LPI mode. †

After the CSR clock is resumed, clear Bit 16 (LPIEN) of Register 12 (LPI Control and Status Register) to bring the MAC out of the LPI mode. †

Programming Guidelines for Flexible Pulse-Per-Second (PPS) Output

Generating a Single Pulse on PPS

To generate single Pulse on PPS: †

1. Program 11 or 10 (for interrupt) in Bits [6:5], TRGTMODSEL, of Register 459 (PPS Control Register) to instruct the MAC to use the Target Time registers (register 455 and 456) for the start time of PPS signal output. †
2. Program the start time value in the Target Time registers (register 455 and 456). †
3. Program the width of the PPS signal output in Register 473 (PPS0 Width Register). †
4. Program Bits [3:0], PPSCMD, of Register 459 (PPS Control Register) to 0001 to instruct the MAC to generate a single pulse on the PPS signal output at the time programmed in the Target Time registers (register 455 and 456). †

Once the PPSCMD is executed (PPSCMD bits = 0), you can cancel the pulse generation by giving the Cancel Start Command (PPSCMD=0011) before the programmed start time elapses. You can also program the behavior of the next pulse in advance. To program the next pulse: †

1. Program the start time for the next pulse in the Target Time registers (register 455 and 456). This time should be more than the time at which the falling edge occurs for the previous pulse. †
2. Program the width of the next PPS signal output in Register 473 (PPS0 Width Register). †
3. Program Bits [3:0], PPSCMD, of Register 459 (PPS Control Register) to generate a single pulse on the PPS signal output after the time at which the previous pulse is de-asserted. And at the time programmed in Target Time registers. If you give this command before the previous pulse becomes low, then the new command overwrites the previous command and the EMAC may generate only 1 extended pulse.

Generating a Pulse Train on PPS

To generate a pulse train on PPS: †

1. Program 11 or 10 (for an interrupt) in Bits [6:5], TRGTMODSEL, of Register 459 (PPS Control Register) to instruct the MAC to use the Target Time registers (register 455 and 456) for the start time of the PPS signal output. †
2. Program the start time value in the Target Time registers (register 455 and 456). †
3. Program the interval value between the train of pulses on the PPS signal output in Register 473 (PPS0 Width Register). †
4. Program the width of the PPS signal output in Register 473 (PPS0 Width Register). †
5. Program Bits[3:0], PPSCMD, of Register 459 (PPS Control Register) to 0010 to instruct the MAC to generate a train of pulses on the PPS signal output with the start time programmed in the Target Time registers (register 455 and 456). By default, the PPS pulse train is free-running unless stopped by 'STOP Pulse train at time' or 'STOP Pulse Train immediately' commands. †
6. Program the stop value in the Target Time registers (register 455 and 456). Ensure that Bit 31 (TSTRBUSY) of Register 456 (Target Time Nanoseconds Register) is clear before programming the Target Time registers (register 455 and 456) again. †
7. Program the PPSCMD field (bit 3:0) of Register 459 (PPS Control Register) to 0100 to stop the train of pulses on the PPS signal output after the programmed stop time specified in step 6 on page 1-71 elapses. †

You can stop the pulse train at any time by programming 0101 in the PPSCMD field. Similarly, you can cancel the Stop Pulse train command (given in step 7 on page 1-71) by programming 0110 in the PPSCMD field before the time (programmed in step 6 on page 1-71) elapses. You can cancel the pulse train generation by programming 0011 in the PPSCMD field before the programmed start time (in step 2 on page 1-71) elapses. †

Generating an Interrupt without Affecting the PPS

Bits [6:5], TRGTMODSEL, of Register 459 (PPS Control Register) enable you to program the Target Time registers (register 455 and 456) to do any one of the following: †

- Generate only interrupts. †
- Generate interrupts and the PPS start and stop time. †
- Generate only PPS start and stop time. †

To program the Target Time registers (register 455 and 456) to generate only interrupt events: †

1. Program 00 (for interrupt) in Bits [6:5], TRGTMODSEL, of Register 459 (PPS Control Register) to instruct the MAC to use the Target Time registers (register 455 and 456) for the target time interrupt. †
2. Program a target time value in the Target Time registers (register 455 and 456) to instruct the MAC to generate an interrupt when the target time elapses. If Bits [6:5], TRGTMODSEL, are changed (for example, to control the PPS), then the interrupt generation is overwritten with the new mode and new programmed Target Time register value.

Ethernet MAC Address Map and Register Definitions

The address map and register definitions pertain to the following modules:

- EMAC Module 0
- EMAC Module 1

Related Information

[Introduction to the Hard Processor System](#) on page 2-1

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) provides two instances of a USB On-The-Go (OTG) controller that supports both device and host functions. The controller supports all high-speed, full-speed, and low-speed transfers in both device and host modes. The controller is fully compliant with the *On The Go and Embedded Host Supplement to the USB Revision 2.0 Specification*. The controller can be programmed for both device and host functions to support data movement over the USB protocol.

The controllers are operationally independent of each other. Each USB OTG controller supports a single USB port connected through a USB 2.0 Transceiver Macrocell Interface Plus (UTMI+) Low Pin Interface (ULPI) compliant PHY. The USB OTG controllers are instances of the Synopsys^(†) DesignWare Cores USB 2.0 Hi-Speed On-The-Go (DWC_otg) controller.

The USB OTG controller is optimized for the following applications and systems: †

- Portable electronic devices †
- Point-to-point applications (no hub, direct connection to HS, FS, or LS device) †
- Multi-point applications (as an embedded USB host) to devices (hub and split support) †

Each of the two USB OTG ports supports both host and device modes, as described in the *On The Go and Embedded Host Supplement to the USB Revision 2.0 Specification*. The USB OTG ports support connections for all types of USB peripherals, including the following peripherals:

- Mouse
- Keyboard
- Digital cameras
- Network adapters
- Hard drives
- Generic hubs

^(†) Portions © 2017 Synopsys, Inc. Used with permission. All rights reserved. Synopsys & DesignWare are registered trademarks of Synopsys, Inc. All documentation is provided "as is" and without any warranty. Synopsys expressly disclaims any and all warranties, express, implied, or otherwise, including the implied warranties of merchantability, fitness for a particular purpose, and non infringement, and any warranties arising out of a course of dealing or usage of trade.

† Paragraphs marked with the dagger (†) symbol are Synopsys Proprietary. Used with permission.

Related Information

<http://www.usb.org/home>

Additional information is available in the On The Go and Embedded Host Supplement to the USB Revision 2.0 Specification, which you can download from the USB Implementers Forum website

Features of the USB OTG Controller

The USB OTG controller has the following USB-specific features:

- Complies with both Revision 1.3 and Revision 2.0 of the *On The Go and Embedded Host Supplement to the USB Revision 2.0 Specification*
- Supports software-configurable modes of operation between OTG 1.3 and OTG 2.0
- Can operate in Host or Device mode
- Supports multi-point applications with hub and split support
- Supports all USB 2.0 speeds:
 - High speed (HS, 480-Mbps)
 - Full speed (FS, 12-Mbps)
 - Low speed (LS, 1.5-Mbps)

Note: In host mode, all speeds are supported. However, in device mode, only high speed and full speed are supported.

- Integrated scatter-gather DMA supports moving data between memory and the controller
- Supports USB 2.0 in ULPI mode
- Supports all USB transaction types:
 - Control transfers
 - Bulk transfers
 - Isochronous transfers
 - Interrupts
- Supports automatic ping capability
- Supports Session Request Protocol (SRP) and Host Negotiation Protocol (HNP)
- Supports suspend, resume, and remote wake
- Supports up to 16 host channels

Note: In host mode, when the number of device endpoints is greater than the number of host channels, software can reprogram the channels to support up to 127 devices, each having 32 endpoints (IN + OUT), for a maximum of 4,064 endpoints.

- Supports up to 16 bidirectional endpoints, including control endpoint 0

Note: Only seven periodic device IN endpoints are supported.

- Supports a generic root hub
- Performs transaction scheduling in hardware

On the USB PHY layer, the USB OTG controller supports the following features:

- 8-bit ULPI PHY data width
- A single USB port connected to each OTG instance
- A ULPI connection to an off-chip USB transceiver
- Software-controlled access, supporting vendor-specific or optional PHY registers access to ease debug
- The OTG 2.0 support for Attach Detection Protocol (ADP) only through an external (off-chip) ADP controller

On the integration side, the USB OTG controller supports the following features:

- Different clocks for system and PHY interfaces
- Dedicated TX FIFO buffer for each device IN endpoint in direct memory access (DMA) mode
- Packet-based, dynamic FIFO memory allocation for endpoints for small FIFO buffers and flexible, efficient use of RAM that can be dynamically sized by software
- Ability to change an endpoint's FIFO memory size during transfers
- Clock gating support during USB suspend and session-off modes
 - PHY clock gating support
 - System clock gating support
- Data FIFO RAM clock gating support
- Local buffering with error correction code (ECC) support

Note: The USB OTG controller does not support the following protocols:

- Enhanced Host Controller Interface (EHCI)
- Open Host Controller Interface (OHCI)
- Universal Host Controller Interface (UHCI)

Supported PHYS

The USB OTG controller only supports USB 2.0 ULPI PHYs. Only the single data rate (SDR) mode is supported.

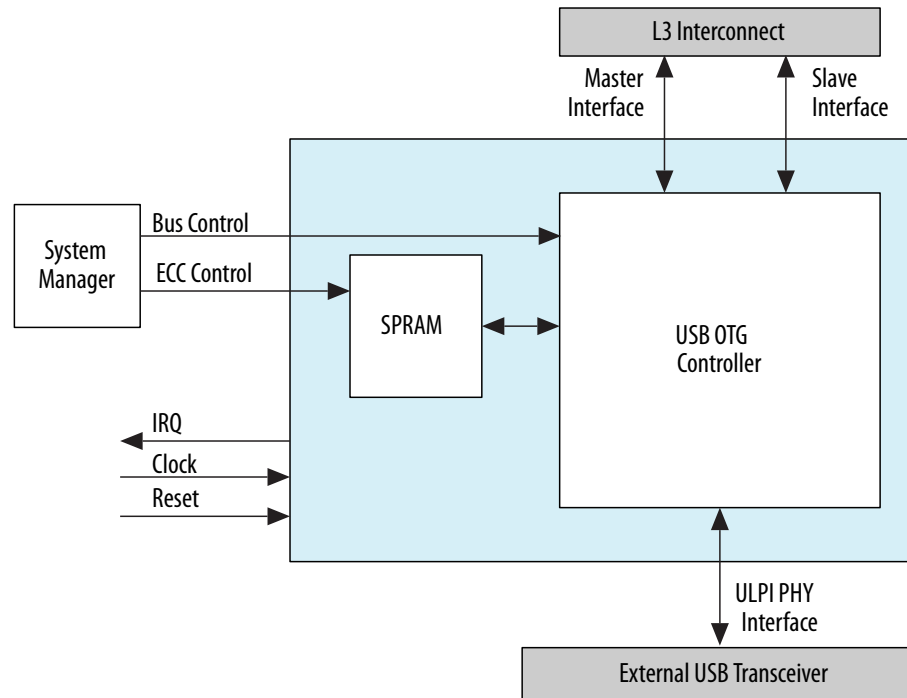
PHYs that support LPM mode may not function properly with the USB controller due to a timing issue. It is recommended that designers use the MicroChip USB3300 PHY device that has been proven to be successful on the development board.

Refer to the *Arria V Device Datasheet* for specific timing information.

USB OTG Controller Block Diagram and System Integration

Figure 18-1: USB OTG Controller System Integration

Two subsystems are included in the HPS.



The USB OTG controller connects to the level 3 (L3) interconnect through a slave interface, allowing other masters to access the control and status registers (CSRs) in the controller. The controller also connects to the L3 interconnect through a master interface, allowing the DMA engine in the controller to move data between external memory and the controller.

A single-port RAM (SPRAM) connected to the USB OTG controller is used to store USB data packets for both host and device modes. It is configured as FIFO buffers for receive and transmit data packets on the USB link.

Through the system manager, the USB OTG controller has control to use and test error correction codes (ECCs) in the SPRAM. Through the system manager, the USB OTG controller can also control the behavior of the master interface to the L3 interconnect.

The USB OTG controller connects to the external USB transceiver through a ULPI PHY interface. This interface also connects through pin multiplexers within the HPS. The pin multiplexers are controlled by the system manager.

Additional connections on the USB OTG controller include:

- Clock input from the clock manager to the USB OTG controller
- Reset input from the reset manager to the USB OTG controller
- Interrupt line from the USB OTG controller to the microprocessor unit (MPU) global interrupt controller (GIC).

The USB controller only uses Direct Shared IO 48.

Related Information

- [System Manager](#) on page 6-1
Details available in the System Manager chapter.
- [General-Purpose I/O Interface](#) on page 22-1

USB 2.0 ULPI PHY Signal Description

Table 18-1: ULPI PHY Interfaces

The ULPI PHY interface is synchronous to the `ulpi_clk` signal coming from the PHY.

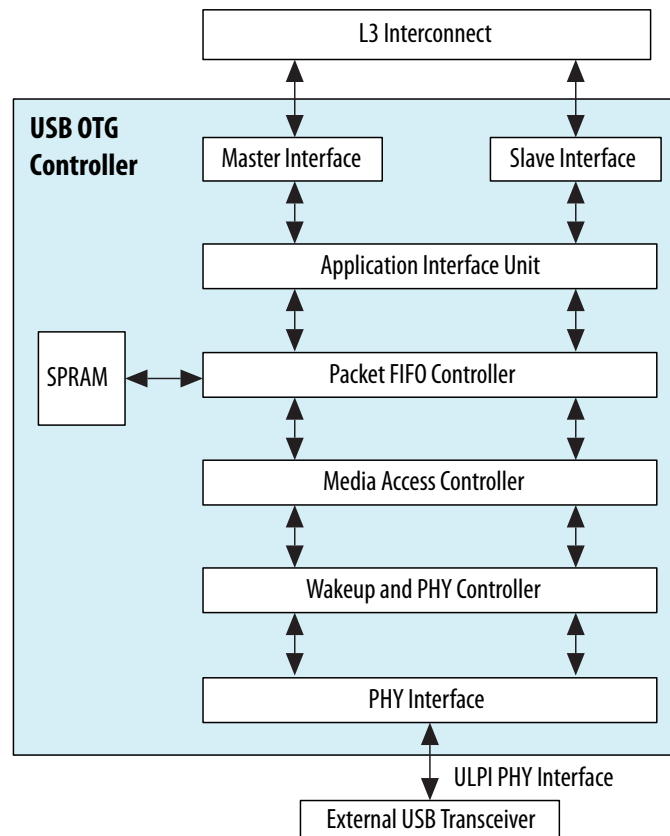
Port Name	Bit Width	Direction	Description
<code>ulpi_clk</code>	1	Input	ULPI Clock Receives the 60-MHz clock supplied by the high-speed ULPI PHY. All signals are synchronous to the positive edge of the clock.
<code>ulpi_dir</code>	1	Input	ULPI Data Bus Control 1—The PHY has data to transfer to the USB OTG controller. 0—The PHY does not have data to transfer.
<code>ulpi_nxt</code>	1	Input	ULPI Next Data Control Indicates that the PHY has accepted the current byte from the USB OTG controller. When the PHY is transmitting, this signal indicates that a new byte is available for the controller.
<code>ulpi_stp</code>	1	Output	ULPI Stop Data Control The controller drives this signal high to indicate the end of its data stream. The controller can also drive this signal high to request data from the PHY.
<code>ulpi_data[7:0]</code>	8	Bidirectional	Bidirectional data bus. Driven low by the controller during idle.

Functional Description of the USB OTG Controller

USB OTG Controller Block Description

Figure 18-2: USB OTG Controller Block Description

Details about each of the units that comprise the USB OTG controller are shown below.



Master Interface

The master interface includes a built-in DMA controller. The DMA controller moves data between external memory and the media access controller (MAC).

Properties of the master interface are controlled through the USB L3 Master HPROT Register (`l3master`) in the system manager. These bits provide access information to the L3 interconnect, including whether or not transactions are cacheable, bufferable, or privileged.

Note: Bits in the `l3master` register can be updated only when the master interface is guaranteed to be in an inactive state.

Slave Interface

The slave interface allows other masters in the system to access the USB OTG controller's CSRs. For testing purposes, other masters can also access the single port RAM (SPRAM).

Slave Interface CSR Unit

The slave interface can read from and write to all the CSRs in the USB OTG controllers. All register accesses are 32 bits.

The CSR is divided into the following groups of registers:

- Global
- Host
- Device
- Power and clock gating

Some registers are shared between host and device modes, because the controller can only be in one mode at a time. The controller generates a mode mismatch interrupt if a master attempts to access device registers when the controller is in host mode, or attempts to access host registers when the controller is in device mode. Writing to unimplemented registers is ignored. Reading from unimplemented registers returns indeterminate values.

Application Interface Unit

The application interface unit (AIU) generates DMA requests based on programmable FIFO buffer thresholds. The AIU generates interrupts to the GIC for both host and device modes. A DMA scheduler is included in the AIU to arbitrate and control the data transfer between packets in system memory and their respective USB endpoints.

Packet FIFO Controller

The Packet FIFO Controller (PFC) connects the AIU with the MAC through data FIFO buffers located in the SPRAM. In device mode, one FIFO buffer is implemented for each IN endpoint. In host mode, a single FIFO buffer stores data for all periodic (isochronous and interrupt) OUT endpoints, and a single FIFO buffer is used for nonperiodic (control and bulk) OUT endpoints. Host and device mode share a single receive data FIFO buffer.

SPRAM

An SPRAM implements the data FIFO buffers for host and device modes. The size of the FIFO buffers can be programmed dynamically.

The SPRAM supports ECCs. ECCs can be enabled through the system manager, by setting the RAM ECC Enable (en) bit in the USB0 or USB1 RAM ECC Enable Register (usb0 or usb1), in the ECC Management Register Group (eccgrp). Single-bit and double-bit errors in each USB instance can be injected using this register.

The SPRAM provides outputs to notify the system manager when single-bit correctable errors are detected (and corrected), and when double-bit (uncorrectable) errors are detected. The system manager generates an interrupt to the GIC when an ECC error is detected.

MAC

The MAC module implements the following functionality:

- USB transaction support
- Host protocol support
- Device protocol support
- OTG protocol support

USB Transactions

In device mode, the MAC decodes and checks the integrity of all token packets. For valid OUT or SETUP tokens, the following DATA packet is also checked. If the data packet is valid, the MAC performs the following steps:

1. Writes the data to the receive FIFO buffer
2. Sends the appropriate handshake when required to the USB host

If a receive FIFO buffer is not available, the MAC sends a NAK response to the host. The MAC also supports ping protocol.

For IN tokens, if data is available in the transmit FIFO buffer, the MAC performs the following steps:

1. Reads the data from the FIFO buffer
2. Forms the data packet
3. Transmits the packet to the host
4. Receives the response from the host
5. Sends the updated status to the PFC

In host mode, the MAC receives a token request from the AIU. The MAC performs the following steps:

1. Builds the token packet
2. Sends the packet to the device

For OUT or SETUP transactions, the MAC also performs the following steps:

1. Reads the data from the transmit FIFO buffer
2. Assembles the data packet
3. Sends the packet to the device
4. Waits for a response

The response from the device causes the MAC to send a status update to the AIU.

For IN or PING transactions, the MAC waits for the data or handshake response from the device. For data responses, the MAC performs the following steps:

1. Validates the data
2. Writes the data to the receive FIFO buffer
3. Sends a status update to the AIU
4. Sends a handshake to the device, if appropriate

Host Protocol

In host mode, the MAC performs the following functions:

- Detects connect, disconnect, and remote wakeup events on the USB link
- Initiates reset
- Initiates speed enumeration processes
- Generates Start of Frame (SOF) packets.

Device Protocol

In device mode, the MAC performs the following functions:

- Handles USB reset sequence
- Handles speed enumeration
- Detects USB suspend and resume activity on the USB link
- Initiates remote wakeup
- Decodes SOF packets

OTG Protocol

The MAC handles HNP and SRP for OTG operation. HNP provides a mechanism for swapping host and device roles. SRP provides mechanisms for the host to turn off V_{BUS} to save power, and for a device to request a new USB session.

Wakeup and Power Control

To reduce power, the USB OTG controller supports a power-down mode. In power-down mode, the controller and the PHY can shut down their clocks. The controller supports wakeup on the detection of the following events:

- Resume
- Remote wakeup
- Session request protocol
- New session start

PHY Interface Unit

The USB OTG controller supports synchronous SDR data transmission to a ULPI PHY. The SDR mode implements an eight-bit data bus.

DMA

The DMA has two modes of operation. You program your software to select between scatter-gather DMA mode or buffer DMA mode depending on the controllers function.

If the controller is functioning as a generic root hub, you should program your software to select the buffer DMA mode that supports split transfers.

If generic root hub functionality is not required, or if the controller is configured in Device mode, you can program your software to select scatter-gather DMA mode.

If you wish to dynamically switch the mode of operation based on the queried device status or capability, your software driver must cleanly switch between the two modes of operation. For example, you may want the controller to default to scatter-gather DMA mode and only change mode when it detects a generic HUB with fast-speed and low-speed capability. Some basic requirements for switching include:

- A soft reset must be issued before changing modes.
- If buffer DMA mode is selected, then the Host mode periodic request queue depth must not be set to 16.
- Devices must be re-enumerated.

Local Memory Buffer

The USB controller has three local SRAM memory buffers.

- The write FIFO buffer is a 128×32 -bit memory (512 total bytes)
- The read FIFO buffer is a 32×32 -bit memory (128 total bytes)
- The ECC buffer is a 96×16 -bit memory (1536 total bytes)

The SPRAM is a 8192×35 -bit (32 data bits and 3 control bits) memory and includes support for ECC (Error Checking and Correction). The ECC block is integrated around a memory wrapper. It provides outputs to notify the system manager when single-bit correctable errors are detected (and corrected) and when double-bit uncorrectable errors are detected. The ECC logic also allows the injection of single- and double-bit errors for test purposes. The ECC feature is disabled by default. It must be initialized to enable the ECC function.

Clocks

Table 18-2: USB OTG Controller Clock Inputs

All clocks must be operational when reset is released. No special handling is required on the clocks.

Clock Signal	Frequency	Functional Usage
usb_mp_clk	60 – 200 MHz	Drives the master and slave interfaces, DMA controller, and internal FIFO buffers
usb0_ulpi_clk	60 MHz	ULPI reference clock for usb0 from external ULPI PHY I/O pin
usb1_ulpi_clk	60 MHz	ULPI reference clock for usb1 from external ULPI PHY I/O pin

Resets

The USB OTG controller can be reset either through the hardware reset input or through software.

Reset Requirements

There must be a minimum of 12 cycles on the `ulpi_clk` clock before the controller is taken out of reset. During reset, the USB OTG controller asserts the `ulpi_stp` signal. The PHY outputs a clock when it sees the `ulpi_stp` signal asserted. However, if the pin multiplexers are not programmed, the PHY does not see the `ulpi_stp` signal. As a result, the `ulpi_clk` clock signal does not arrive at the USB OTG controller.

Software must ensure that the reset is active for a minimum of two `usb_mp_clk` cycles. There is no maximum assertion time.

Hardware Reset

Each of the USB OTG controllers has one reset input from the reset manager. The reset signal is asserted during a cold or warm reset event. The reset manager holds the controllers in reset until software releases the resets. Software releases resets by clearing the appropriate USB bits in the Peripheral Module Reset Register (`permodrst`) in the HPS reset manager.

The reset input resets the following blocks:

- The master and slave interface logic
- The integrated DMA controller
- The internal FIFO buffers
- The CSR

The reset input is synchronized to the `usb_mp_clk` domain. The reset input is also synchronized to the ULPI clock within the USB OTG controller and is used to reset the ULPI PHY domain logic.

Software Reset

Software can reset the controller by setting the Core Soft Reset (`csftrst`) bit in the Reset Register (`grstctl`) in the Global Registers (`globgrp`) group of the USB OTG controller.

Software resets are useful in the following situations:

- A PHY selection bit is changed by software. Resetting the USB OTG controller is part of clean-up to ensure that the PHY can operate with the new configuration or clock.
- During software development and debugging.

Taking the USB 2.0 OTG Controller Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

Interrupts

Table 18-3: USB OTG Interrupt Conditions

Each USB OTG controller has a single interrupt output. Interrupts are asserted on the conditions shown in the following table.

Condition	Mode
Device-initiated remote wakeup is detected.	Host mode

Condition	Mode
Session request is detected from the device.	Host mode
Device disconnect is detected.	Host mode
Host periodic TX FIFO buffer is empty (can be further programmed to indicate half-empty).	Host mode
Host channels interrupt received.	Host mode
Incomplete periodic transfer is pending at the end of the microframe.	Host mode
Host port status interrupt received.	Host mode
External host initiated resume is detected.	Device mode
Reset is detected when in suspend or normal mode.	Device mode
USB suspend mode is detected.	Device mode
Data fetch is suspended due to TX FIFO buffer full or request queue full.	Device mode
At least one isochronous OUT endpoint is pending at the end of the microframe.	Device mode
At least one isochronous IN endpoint is pending at the end of the microframe.	Device mode
At least one IN or OUT endpoint interrupt is pending at the end of the microframe.	Device mode
The end of the periodic frame is reached.	Device mode
Failure to write an isochronous OUT packet to the RX FIFO buffer. The RX FIFO buffer does not have enough space to accommodate the maximum packet size for the isochronous OUT endpoint.	Device mode
Enumeration has completed.	Device mode
Connector ID change.	Common modes
Mode mismatch. Software accesses registers belonging to an incorrect mode.	Common modes
Nonperiodic TX FIFO buffer is empty.	Common modes
RX FIFO buffer is not empty.	Common modes

Condition	Mode
Start of microframe.	Common modes
Device connection debounce is complete in host mode.	OTG interrupts
A-Device timeout while waiting for B-Device connection.	OTG interrupts
Host negotiation is complete.	OTG interrupts
Session request is complete.	OTG interrupts
Session end is detected in device mode.	OTG interrupts

USB OTG Controller Programming Model

For detailed information about using the USB OTG controller, consult your operating system (OS) driver documentation. The OS vendor provides application programming interfaces (APIs) to control USB host, device and OTG operation. This section provides a brief overview of the following software operations:

- Enabling SPRAM ECCs
- Host operation
- Device operation

Enabling ECC

1. Turn on the ECC hardware, but disable the interrupts.
2. Initialize the SRAM in the peripheral.⁽⁵⁵⁾
3. Clear the ECC event bits, because these bits may have become asserted after Step 1.
4. Enable the ECC interrupts now that the ECC bits have been set.

Enabling SPRAM ECCs

To avoid false ECC errors, you must initialize the ECC bits in the SPRAM before using ECCs. To initialize the ECC bits, software writes data to all locations in the SPRAM.

The L3 interconnect has access to the SPRAM and is accessible through the USB OTG L3 slave interface. Software accesses the SPRAM through the `directfifo` memory space, in the USB OTG controller address space.

The SPRAM contains 8192 (32 KB) locations. The L3 slave provides 32-bit access to the SPRAM. Physically, the SPRAM is implemented as a 35-bit memory, with the highest three bits reserved for the USB OTG controller's internal use. When a write is performed to the SPRAM through the L3 slave interface, bits 32 through 34 of the internal data bus are tied to 1, to enable the ECC bits to be initialized.

Note: Software cannot access the SPRAM beyond the 32-KB range. Out-of-range read transactions return indeterminate data. Out-of-range write transactions are ignored.

⁽⁵⁵⁾ This is the case for the following peripherals: SD/MMC, NAND, On-Chip, DMA, USB, and EMAC.

Host Operation

Host Initialization

After power up, the USB port is in its default mode. No VBUS is applied to the USB cable. The following process sets up the USB OTG controller as a USB host.

1. To enable power to the USB port, the software driver sets the Port Power (`prt pwr`) bit to 1 in the Host Port Control and Status Register (`hp rt`) of the Host Mode Registers (`hostgrp`) group. This action drives the V_{BUS} signal on the USB link.

The controller waits for a connection to be detected on the USB link.

2. When a USB device connects, an interrupt is generated. The Port Connect Detected (`PrtConnDet`) bit in `hp rt` is set to 1.
3. Upon detecting a port connection, the software driver initiates a port reset by setting the Port Reset (`prt rst`) bit to 1 in `hp rt`.
4. The software driver must wait a minimum of 10 ms so that speed enumeration can complete on the USB link.
5. After the 10 ms, the software driver sets `prt rst` back to 0 to release the port reset.
6. The USB OTG controller generates an interrupt. The Port Enable Disable Change (`prt enchng`) and Port Speed (`prt spd`) bits, in `hp rt`, are set to reflect the enumerated speed of the device that attached.

At this point the port is enabled for communication. Keep alive or SOF packets are sent on the port. If a USB 2.0-capable device fails to initialize correctly, it is reported as a USB 1.1 device.

The Host Frame Interval Register (`hf ir`) is updated with the corresponding PHY clock settings. The `hf ir`, used for sending SOF packets, is in the Host Mode Registers (`hostgrp`) group.

7. The software driver must program the following registers in the Global Registers (`globgrp`) group, in the order listed:
 - a. Receive FIFO Size Register (`grxf siz`)—selects the size of the receive FIFO buffer
 - b. Non-periodic Transmit FIFO Size Register (`gnptxf siz`)—selects the size and the start address of the non-periodic transmit FIFO buffer for nonperiodic transactions
 - c. Host Periodic Transmit FIFO Size Register (`hptxf siz`)—selects the size and start address of the periodic transmit FIFO buffer for periodic transactions
8. System software initializes and enables at least one channel to communicate with the USB device.

Host Transaction

When configured as a host, the USB OTG controller pipes the USB transactions through one of two request queues (one for periodic transactions and one for nonperiodic transactions). Each entry in the request queue holds the SETUP, IN, or OUT channel number along with other information required to perform a transaction on the USB link. The sequence in which the requests are written to the queue determines the sequence of transactions on the USB link.

The host processes the requests in the following order at the beginning of each frame or microframe:

1. Periodic request queue, including isochronous and interrupt transactions
2. Nonperiodic request queue (bulk or control transfers)

The host schedules transactions for each enabled channel in round-robin fashion. When the host controller completes the transfer for a channel, the controller updates the DMA descriptor status in the system memory.

For OUT transactions, the host controller uses two transmit FIFO buffers to hold the packet payload to be transmitted. One transmit FIFO buffer is used for all nonperiodic OUT transactions and the other is used for all periodic OUT transactions.

For IN transactions, the USB host controller uses one receive FIFO buffer for all periodic and nonperiodic transactions. The controller holds the packet payload from the USB device in the receive FIFO buffer until the packet is transferred to the system memory. The receive FIFO buffer also holds the status of each packet received. The status entry holds the IN channel number along with other information, including received byte count and validity status.

For generic hub operations, the USB OTG controller uses SPLIT transfers to communicate with slower-speed devices downstream of the hub. For these transfers, the transaction accumulation or buffering is performed in the generic hub, and is scheduled accordingly. The USB OTG controller ensures that enough transmit and receive buffers are allocated when the downstream transactions are completed or when accumulated data is ready to be sent upstream.

Device Operation

Device Initialization

The following process sets up the USB OTG controller as a USB device:

1. After power up, the USB OTG controller must be set to the desired device speed by writing to the Device Speed (`devspd`) bits in the Device Configuration Register (`dcfg`) in the Device Mode Registers (`devgrp`) group. After the device speed is set, the controller waits for a USB host to detect the USB port as a device port.
2. When an external host detects the USB port, the host performs a port reset, which generates an interrupt to the USB device software. The USB Reset (`usbrst`) bit in the Interrupt (`port_reset`) register in the Global Registers (`globgrp`) group is set. The device software then sets up the data FIFO buffer to receive a SETUP packet from the external host. Endpoint 0 is not enabled yet.
3. After completion of the port reset, the operation speed required by the external host is known. Software reads the device speed status and sets up all the remaining required transaction fields to enable control endpoint 0.

After completion of this process, the device is receiving SOF packets, and is ready for the USB host to set up the device's control endpoint.

Device Transaction

When configured as a device, the USB OTG controller uses a single FIFO buffer to receive the data for all the OUT endpoints. The receive FIFO buffer holds the status of the received data packet, including the byte count, the data packet ID (PID), and the validity of the received data. The DMA controller reads the data out of the FIFO buffer as the data are received. If a FIFO buffer overflow condition occurs, the controller responds to the OUT packet with a NAK, and internally rewinds the pointers.

For IN endpoints, the controller uses dedicated transmit buffers for each endpoint. The application does not need to predict the order in which the USB host accesses the nonperiodic endpoints. If a FIFO buffer underrun condition occurs during transmit, the controller inverts the cyclic redundancy code (CRC) to mark the packet as corrupt on the USB link.

The application handles one data packet at a time per endpoint in transaction-level operations. The software receives an interrupt on completion of every packet. Based on the handshake response received

on the USB link, the application determines whether to retry the transaction or proceed with the next transaction, until all packets in the transfer are completed.

When an isochronous IN endpoint has more than one transaction (or packet) per micro-frame, it is referred to as a high-bandwidth isochronous endpoint. This endpoint should follow the Data PID sequencing specified in section 5.9.2 of the USB 2.0 Specification. This allows the device controller to support a maximum of three transactions per micro-frame and high-bandwidth isochronous IN endpoints in the Ignore Frame Number mode of the device descriptor DMA configuration.

IN Transactions

For an IN transaction, the application performs the following steps:

1. Enables the endpoint
2. Triggers the DMA engine to write the associated data packet to the corresponding transmit FIFO buffer
3. Waits for the packet completion interrupt from the controller

When an IN token is received on an endpoint when the associated transmit FIFO buffer does not contain sufficient data, the controller performs the following steps:

1. Generates an interrupt
2. Returns a NAK handshake to the USB host

If sufficient data is available, the controller transmits the data to the USB host.

OUT Transactions

For an OUT transaction, the application performs the following steps:

1. Enables the endpoint
2. Waits for the packet received interrupt from the USB OTG controller
3. Retrieves the packet from the receive FIFO buffer

When an OUT token or PING token is received on an endpoint where the receive FIFO buffer does not have sufficient space, the controller performs the following steps:

1. Generates an interrupt
2. Returns a NAK handshake to USB host

If sufficient space is available, the controller stores the data in the receive FIFO buffer and returns an ACK handshake to the USB link.

According to the USB 2.0 specifications, to transmit two packets back-to-back, the inter-packet delay must be a minimum of 88 bit times, measured at the first receptacle of the host. This guarantees an inter-packet delay of at least 32 bit times at all devices (when receiving these back-to-back packets). For the device controller to support the worst-case inter-packet delay of 32 bit times and isochronous OUT endpoints across multiple layers of hubs, the controller must operate in 8-bit UTMI mode and the AHB frequency must be at least 96 MHz.

Alternatively, choose the AHB clock frequency based on your targeted use case and supported number of hubs. However, in both cases the 8-bit PHY data interface should be used for operation using the SoC.

Table 18-4: Supported Number of Hubs Tiers for Different PHY and AHB Clock Frequencies

Number of Hub Levels	Worst Case IPG Possible at Hub/Host Downstream Port (in bit times)	Minimum HCLK Frequency Required (in MHz)	
		UTMI Data Width = 8	UTMI Data Width = 16
No Hub	88	30	110
1	78.4	32	213
2	68.8	37	-
3	59.2	44	-
4	49.6	54	-
5	40	70	-

Control Transfers

For control transfers, the application performs the following steps:

1. Waits for the packet received interrupt from the controller
2. Retrieves the packet from the receive buffer

Because the control transfer is governed by USB protocol, the controller always responds with an ACK handshake.

USB 2.0 OTG Controller Address Map and Register Definitions

The address map and register definitions for the USB OTG Controller consists of the following region:

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
The base addresses of all modules are listed in the *Introduction to the Hard Processor System* chapter.
- <https://www.intel.com/content/www/us/en/programmable/hps/arria-v/hps.html>

USB Data FIFO Address Map

These regions, available in both Host and Device modes, are a push/pop FIFO space for a specific endpoint or a channel, in a given direction. If a host channel is of type IN, the FIFO can only be read on the channel. Similarly, if a host channel is of type OUT, the FIFO can only be written on the channel.

Module Instance	Base Address
usb0	0xFFB00000
usb1	0xFFB40000

Table 18-5: USB Endpoint FIFO Address Ranges

Name	Description	Start Address Offset	End Address Offset
EP0/HC0 FIFO	This address space is allocated for Endpoint 0/Host Channel 0 push/pop FIFO access.	0x1000	0x1FFF
EP1/HC1 FIFO	This address space is allocated for Endpoint 1/Host Channel 1 push/pop FIFO access.	0x2000	0x2FFF
EP2/HC2 FIFO	This address space is allocated for Endpoint 2/Host Channel 2 push/pop FIFO access.	0x3000	0x3FFF
EP3/HC3 FIFO	This address space is allocated for Endpoint 3/Host Channel 3 push/pop FIFO access.	0x4000	0x4FFF
EP4/HC4 FIFO	This address space is allocated for Endpoint 4/Host Channel 4 push/pop FIFO access.	0x5000	0x5FFF
EP5/HC5 FIFO	This address space is allocated for Endpoint 5/Host Channel 5 push/pop FIFO access.	0x6000	0x6FFF
EP6/HC6 FIFO	This address space is allocated for Endpoint 6/Host Channel 6 push/pop FIFO access.	0x7000	0x7FFF
EP7/HC7 FIFO	This address space is allocated for Endpoint 7/Host Channel 7 push/pop FIFO access.	0x8000	0x8FFF
EP8/HC8 FIFO	This address space is allocated for Endpoint 8/Host Channel 8 push/pop FIFO access.	0x9000	0x9FFF
EP9/HC9 FIFO	This address space is allocated for Endpoint 9/Host Channel 9 push/pop FIFO access.	0xA000	0xAFFF

Name	Description	Start Address Offset	End Address Offset
EP10/HC10 FIFO	This address space is allocated for Endpoint 10/Host Channel 10 push/pop FIFO access.	0xB000	0xBFFF
EP11/HC11 FIFO	This address space is allocated for Endpoint 11/Host Channel 11 push/pop FIFO access.	0xC000	0xCFFF
EP12/HC12 FIFO	This address space is allocated for Endpoint 12/Host Channel 12 push/pop FIFO access.	0xD000	0xDFFF
EP13/HC13 FIFO	This address space is allocated for Endpoint 13/Host Channel 13 push/pop FIFO access.	0xE000	0xEFFF
EP14/HC14 FIFO	This address space is allocated for Endpoint 14/Host Channel 14 push/pop FIFO access.	0xF000	0xFFFF
EP15/HC15 FIFO	This address space is allocated for Endpoint 15/Host Channel 15 push/pop FIFO access.	0x10000	0x10FFF

USB Direct Access FIFO RAM Address Map

This address space provides direct access to the Data FIFO RAM for debugging.

Module Instance	Base Address
usb0	0xFFB00000
usb1	0xFFB40000

Table 18-6: USB Direct Access FIFO Address Range

Name	Description	Start Address Offset	End Address Offset
Direct_FIFO	This address space is allocated for directly accessing the data FIFO for debugging purposes.	0x20000	0x3FFFF

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) provides two serial peripheral interface (SPI) masters and two SPI slaves. The SPI masters and slaves are instances of the Synopsys DesignWare Synchronous Serial Interface (SSI) controller (DW_apb_ssi). †⁽⁵⁶⁾

Features of the SPI Controller

The SPI controller has the following features: †

- Serial master and serial slave controllers – Enable serial communication with serial-master or serial-slave peripheral devices. †
- Serial interface operation – Programmable choice of the following protocols:
 - Motorola SPI protocol
 - Texas Instruments Synchronous Serial Protocol
 - National Semiconductor Microwire
- DMA controller interface integrated with HPS DMA controller
- SPI master supports received serial data bit (RXD) sample delay
- Transmit and receive FIFO buffers are 256 words deep
- SPI master supports up to four slave selects
- Programmable master serial bit rate
- Programmable data item size of 4 to 16 bits
- Support for Multi-master mode through the FPGA logic

⁽⁵⁶⁾ Portions © 2017 Synopsys, Inc. Used with permission. All rights reserved. Synopsys & DesignWare are registered trademarks of Synopsys, Inc. All documentation is provided "as is" and without any warranty. Synopsys expressly disclaims any and all warranties, express, implied, or otherwise, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, and any warranties arising out of a course of dealing or usage of trade.

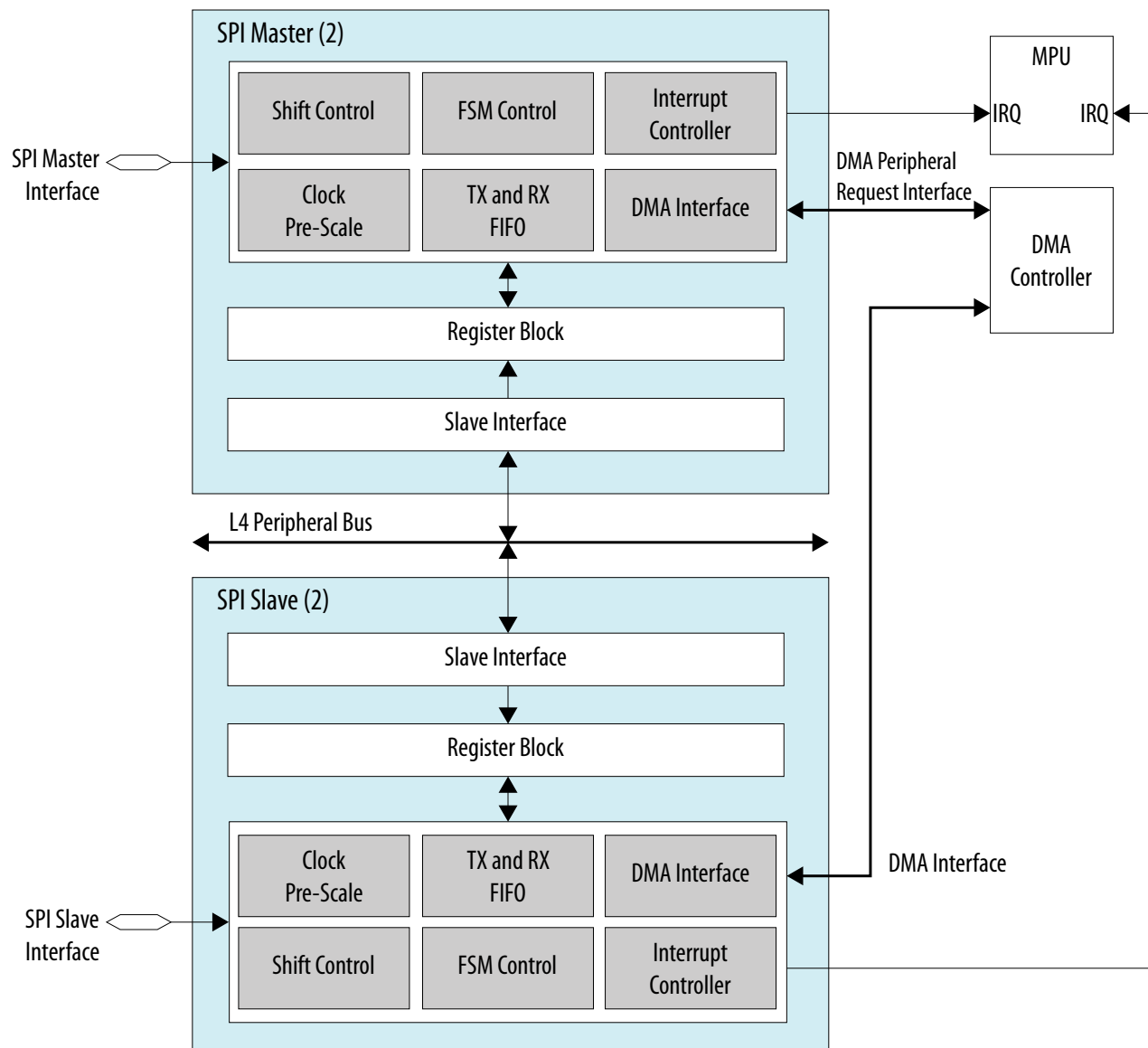
†Paragraphs marked with the dagger (†) symbol are Synopsys Proprietary. Used with permission.

SPI Block Diagram and System Integration

The SPI supports data bus widths of 32 bits. †

SPI Block Diagram

Figure 19-1: SPI Block Diagram



The functional groupings of the main interfaces to the SPI block are as follows: †

- System bus interface
- DMA peripheral request interface
- Interrupt interface
- SPI interface

SPI Controller Signal Description

Signals from the two SPI masters and two SPI slaves can be routed to the FPGA or the HPS I/O pins. The following sections describe the signals available.

Interface to HPS I/O

Table 19-1: SPI Master Interface Pins

Signal Name	Signal Width	Direction	Description
CLK	1	Out	Serial clock output from the SPI master
MOSI	1	Out	Transmit data line for the SPI master
MISO	1	In	Receive data line for the SPI master
SS0	1	Out	Slave Select 0 Slave select signal from SPI master
SS1	1	Out	Slave Select 1 Slave select signal from SPI master

Table 19-2: SPI Slave Interface Pins

CLK	1	In	Serial clock input to the SPI slave
MOSI	1	In	Receive data line for the SPI slave
MISO	1	Out	Transmit data line for the SPI slave
SS0	1	In	Slave select input to the SPI slave

FPGA Routing

Table 19-3: SPI Master Signals for FPGA routing:

Signal Name	Signal Width	Direction	Description
spim_txd	1	Out	Transmit data line for the SPI master
spim_rxd	1	In	Receive data line for the SPI master



Signal Name	Signal Width	Direction	Description
spim_ss_in_n	1	In	Master Contention Input
spim_ssi_oe_n	1	Out	Output enable for the SPI master
spim_ss_0_n	1	Out	Slave Select 0 Slave select signal from SPI master
spim_ss_1_n	1	Out	Slave Select 1 Allows second slave to be connected to this master
spim_ss_2_n	1	Out	Slave Select 2 Allows third slave to be connected to this master
spim_ss_3_n	1	Out	Slave Select 3 Allows fourth slave to be connected to this master

Table 19-4: SPI Slave Signals for FPGA Routing

spis_txd	1	Out	Transmit data line for the SPI slave
spis_rxd	1	In	Receive data line for the SPI slave
spis_ss_in_n	1	In	Master Contention Input
spis_ssi_oe_n	1	Out	Output enable for the SPI slave
spis_sclk_in	1	In	Serial clock input

Functional Description of the SPI Controller

Protocol Details and Standards Compliance

This section describes the functional operation of the SPI controller.

The host processor accesses data, control, and status information about the SPI controller through the system bus interface. The SPI also interfaces with the DMA Controller. †

The HPS includes two general-purpose SPI master controllers and two general-purpose SPI slave controllers.

The SPI controller can connect to any other SPI device using any of the following protocols:

- Motorola SPI Protocol †
- Texas Instruments Serial Protocol (SSP) †
- National Semiconductor Microwire Protocol †

SPI Controller Overview

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

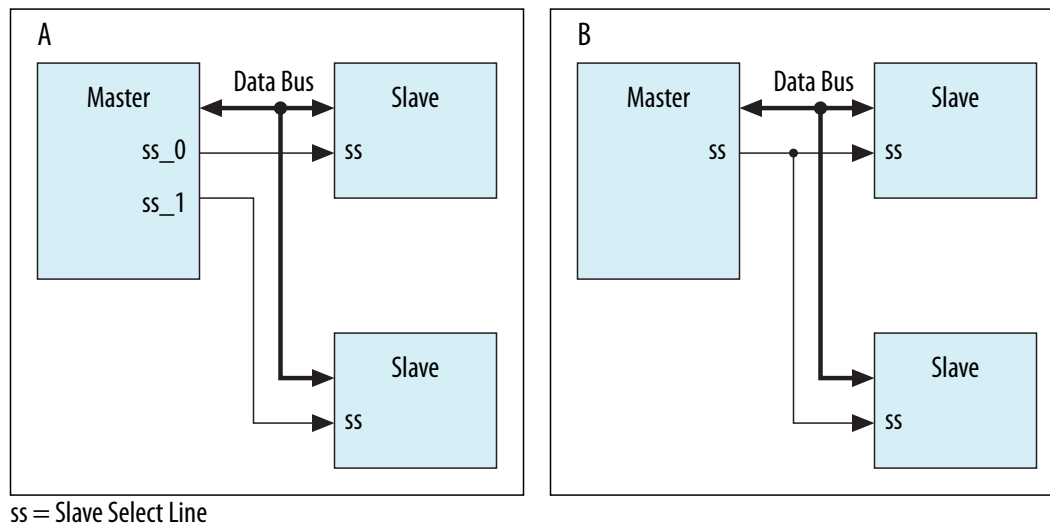
- Motorola SPI protocol – A four-wire, full-duplex serial protocol from Motorola. The slave select line is held high when the SPI controller is idle or disabled. For more information, refer to “Motorola SPI Protocol”. †
- Texas Instruments Serial Protocol (SSP) – A four-wire, full-duplex serial protocol. 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)”. †
- 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 Protocol”. You can program the FRF (frame format) bit field in the Control Register 0 (CTRLR0) to select which protocol is used. †

The serial protocols supported by the SPI controller allow for serial slaves to be selected or addressed using 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 part A of [Figure 19-2](#). †

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 part B of the [Figure 19-2](#) figure below. Software would control which slave is to respond to the serial transfer request from the master device. †

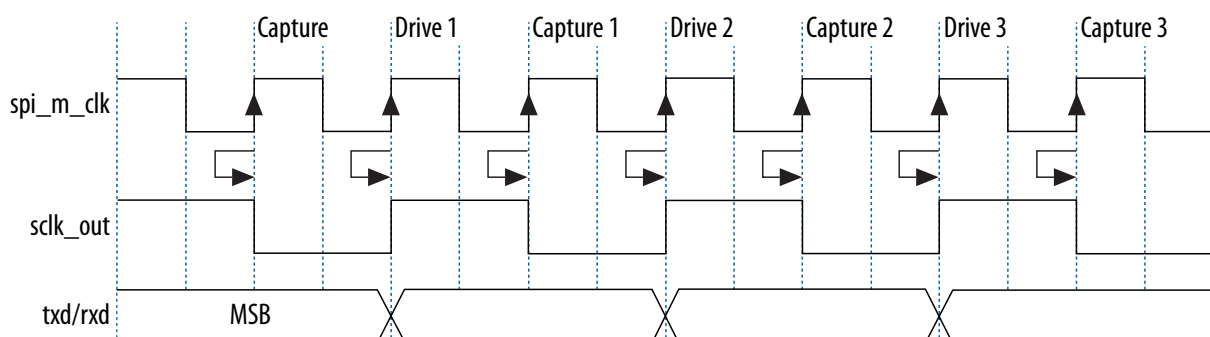
Figure 19-2: Hardware/Software Slave Selection

**Related Information**

- [Motorola SPI Protocol](#) on page 19-16
- [Texas Instruments Synchronous Serial Protocol \(SSP\)](#) on page 19-18
- [National Semiconductor Microwire Protocol](#) on page 19-19

Serial Bit-Rate Clocks**SPI Master Bit-Rate Clock**

The maximum frequency of the SPI master bit-rate clock (`sclk_out`) is one-half the frequency of SPI master clock (`spi_m_clk`). This allows the shift control logic to capture data on one clock edge of `sclk_out` and propagate data on the opposite edge. 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. †

Figure 19-3: Maximum `sclk_out`/`spi_m_clk` Ratio

The frequency of `sclk_out` can be derived from the equation below, where `<SPI clock>` is `spi_m_clk` for the master SPI modules and `l4_main_clk` for the slave SPI modules. †

$$F_{\text{sclk\_out}} = F_{\text{<SPI clock>}} / \text{SCKDV}$$

SCKDV is a bit field in the register BAUDR, holding any even value in the range 2 to 65,534. If SCKDV is 0, then `sclk_out` is disabled. †

The following equation describes the frequency ratio restrictions between the bit-rate clock `sclk_out` and the SPI master peripheral clock. The SPI master peripheral clock must be at least double the offchip master clock. †

Table 19-5: SPI Master Peripheral Clock

SPI Master Peripheral Clock
$F_{\text{spi\_m\_clk}} \geq 2 \times (\text{maximum } F_{\text{sclk\_out}})$ †

SPI Slave Bit-Rate Clock

The minimum frequency of `l4_main_clk` depends on the operation of the slave peripheral. If the slave device is *receive only*, the minimum frequency of `l4_main_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 `l4_main_clk` domain, and then it is edge detected; this synchronization requires three `l4_main_clk` periods. †

If the slave device is *transmit and receive*, the minimum frequency of `l4_main_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 `rx` line is stable before the master shift control logic captures the data. †

The frequency ratio restrictions between the bit-rate clock `sclk_in` and the SPI slave peripheral clock are as follows: †

- Slave (receive only): $F_{\text{l4\_main\_clk}} \geq 6 \times (\text{maximum } F_{\text{sclk\_in}})$ †
- Slave: $F_{\text{l4\_main\_clk}} \geq 8 \times (\text{maximum } F_{\text{sclk\_in}})$ †

Transmit and Receive FIFO Buffers

There are two 16-bit FIFO buffers, a transmit FIFO buffer and a receive FIFO buffer, with a depth of 256. 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 buffer location; for example, you may not store two 8-bit data frames in a single FIFO buffer location. If an 8-bit data frame is required, the upper 8-bits of the FIFO buffer entry are ignored or unused when the serial shifter transmits the data. †

The transmit and receive FIFO buffers are cleared when the SPI controller is disabled (`SSIENR=0`) or reset.

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

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

The threshold value allows you to provide early indication to the processor that the receive FIFO buffer is nearly full. A Receive FIFO Overflow Interrupt is generated when the receive shift logic attempts to load data into a completely full receive FIFO buffer. However, the newly received data are lost. A Receive FIFO Underflow Interrupt is generated if you attempt to read from an empty receive FIFO buffer. This alerts the processor that the read data are invalid. †

Related Information

[Reset Manager](#) on page 4-1

For more information, refer to the *Reset Manager* chapter.

SPI Interrupts

The SPI controller supports combined interrupt requests, which can be masked. The combined interrupt request is the ORed result of all other SPI interrupts after masking. All SPI interrupts have active-high polarity level. The SPI interrupts are described as follows: †

- Transmit FIFO Empty Interrupt – Set when the transmit FIFO buffer is equal to or below its threshold value and requires service to prevent an underrun. The threshold value, set through a software-programmable register, determines the level of transmit FIFO buffer 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 – Set when a master attempts to write data into the transmit FIFO buffer after it has been completely filled. When set, new data writes are discarded. This interrupt remains set until you read the transmit FIFO overflow interrupt clear register (TXOICR). †
- Receive FIFO Full Interrupt – Set when the receive FIFO buffer 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 buffer 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 – Set when the receive logic attempts to place data into the receive FIFO buffer 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 – Set when a system bus access attempts to read from the receive FIFO buffer when it is empty. When set, zeros are read back from the receive FIFO buffer. This interrupt remains set until you read the receive FIFO underflow interrupt clear register (RXUICR). †
- Combined Interrupt Request – ORed result of all the above interrupt requests after masking. To mask this interrupt signal, you must mask all other SPI interrupt requests. †

Transmit FIFO Overflow, Transmit FIFO Empty, Receive FIFO Full, Receive FIFO Underflow, and Receive FIFO Overflow interrupts can all be masked independently, using the Interrupt Mask Register (IMR). †

Transfer Modes

When transferring data on the serial bus, the SPI controller operates one of several modes. The transfer mode (TMOD) is set by writing to the TMOD field in control register 0 (CTRLR0).

Note: 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. †

Transmit and Receive

When TMOD = 0, 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 buffer 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 buffer at the end of each data frame. †

Transmit Only

When TMOD = 1, any receive data are ignored. The data transfer occurs as normal, according to the selected frame format (serial protocol). Transmit data are popped from the transmit FIFO buffer 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 buffer. 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. †

Receive Only

When TMOD = 2, the transmit data are invalid. In the case of the SPI slave, the transmit FIFO buffer 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 buffer at the end of each data frame. You should mask interrupts originating from the transmit logic when this mode is entered. †

EEPROM Read

Note: This transfer mode is only valid for serial masters. †

When TMOD = 3, the transmit data is used to transmit an opcode, an address, or both to the EEPROM device. 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 SPI master is transmitting data on its txd line, data on the rxd line is ignored). The SPI master continues to transmit data until the transmit FIFO buffer is empty. You should ONLY have enough data frames in the transmit FIFO buffer to supply the opcode and address to the EEPROM. If more data frames are in the transmit FIFO buffer than are needed, then Read data is lost.

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

Note: EEPROM read mode is not supported when the SPI controller is configured to be in the SSP mode. †

SPI Master

The SPI master initiates and controls all serial transfers with serial-slave peripheral devices. †

The serial bit-rate clock, generated and controlled by the SPI controller, is driven out on the scl_{k_out} line. When the SPI controller is disabled, no serial transfers can occur and scl_{k_out} is held in “inactive” state, as defined by the serial protocol under which it operates. †

Related Information

[SPI Block Diagram](#) on page 19-2

RXD Sample Delay

The SPI master device is capable of delaying the default sample time of the `rx_d` signal in order to increase the maximum achievable frequency on the serial bus.

Round trip routing delays on the `sclk_out` signal from the master and the `rx_d` signal from the slave can mean that the timing of the `rx_d` signal, as seen by the master, has moved away from the normal sampling time.

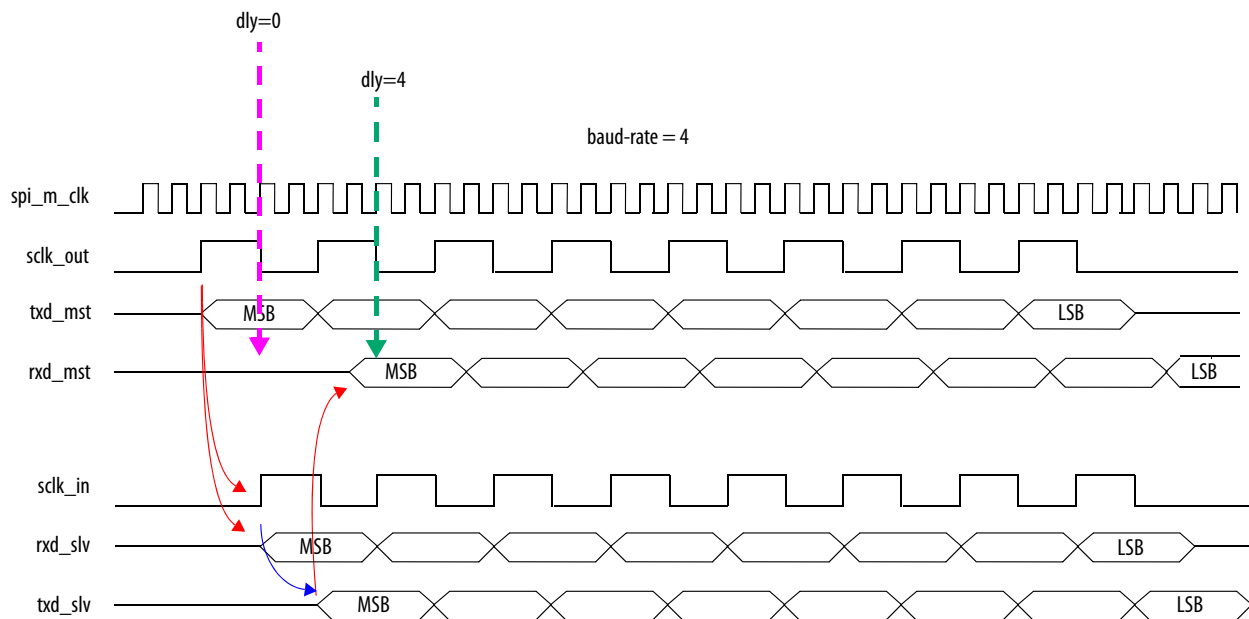
Without the RXD sample delay, you must increase the baud rate for the transfer in order to ensure that the setup times on the `rx_d` signal are within range. This reduces the frequency of the serial interface.

Additional logic is included in the SPI master 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. †

By writing to the `rsd` field of the RXD sample delay register (`rx_sample_dly`), you specify an additional amount of delay applied to the `rx_d` sample. The delay is in number of `l4_main_clk` clock cycles, with 64 maximum cycles allowed (zero is reserved). If the `rsd` field is programmed with a value exceeding 64, a zero delay is applied to the `rx_d` sample.

The sample delay logic has a resolution of one `l4_main_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. †

Figure 19-4: Effects of Round Trip Routing Delays on `sclk_out` Signal



Red arrows indicates routing delay between master and slave devices
Blue arrow indicates sampling delay within slave from receiving `sclk_in` to driving `txd_out`

Data Transfers

The SPI master starts data transfers when all the following conditions are met:

- The SPI master is enabled
- There is at least one valid entry in the transmit FIFO buffer
- A 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 buffer. This bit gets set only when the target slave has been selected and the transfer is underway. After writing data into the transmit FIFO buffer, 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 Transmit FIFO Empty (TFE) status (waiting for 1) or wait for (BAUDR * SPI clock) clock cycles. †

Master SPI and SSP Serial Transfers

“Motorola SPI Protocol” and “Texas Instruments Synchronous Serial Protocol (SSP)” describe the SPI and SSP serial protocols, respectively. †

When the transfer mode is “transmit and receive” or “transmit only” (TMOD = 0 or TMOD = 1, respectively), transfers are terminated by the shift control logic when the transmit FIFO buffer 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 (Transmit FIFO Empty Interrupt) the processor indicating that the transmit FIFO buffer is nearly empty. †

When the DMA is used in conjunction with the SPI master, the transmit data level (DMATDLR) can be used to early request the DMA Controller, indicating that the transmit FIFO buffer is nearly empty. The FIFO buffer can then be refilled with data to continue the serial transfer. The user may also write a block of data (at least two FIFO buffer entries) into the transmit FIFO buffer 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 buffer. †

When the transfer mode is “receive only” (TMOD = 2), a serial transfer is started by writing one “dummy” data word into the transmit FIFO buffer when a serial slave is selected. The `txd` output from the SPI controller is held at a constant logic level for the duration of the serial transfer. The transmit FIFO buffer 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 plus one. This transfer mode increases the bandwidth of the system bus as the transmit FIFO buffer never needs to be serviced during the transfer. The receive FIFO buffer should be read each time the receive FIFO buffer generates a FIFO full interrupt request to prevent an overflow. †

When the transfer mode is “eeprom_read” (TMOD = 3), a serial transfer is started by writing the opcode or address, or both, into the transmit FIFO buffer 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 buffer. 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 SPI controller 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 buffer is nearly full. When a DMA is used, the receive data level (DMARDLR) can be used to early request the DMA Controller, indicating that the receive FIFO buffer is nearly full. †

Related Information

- [Motorola SPI Protocol](#) on page 19-16
- [Texas Instruments Synchronous Serial Protocol \(SSP\)](#) on page 19-18
- [SPI Controller Address Map and Register Definitions](#) on page 19-33

Master Microwire Serial Transfers

“National Semiconductor Microwire Protocol” describes the Microwire serial protocol in detail. †

Microwire serial transfers from the SPI serial master are controlled by the Microwire Control Register (MWCR). The MHS 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 SPI serial master when there is at least one control word in the transmit FIFO buffer and a slave is enabled. When the SPI master transmits the data frame (MDD = 1), the transfer is terminated by the shift logic when the transmit FIFO buffer is empty. When the SPI 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 buffer 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 plus one. †

When the handshaking interface on the SPI master is enabled (MHS = 1), the status of the target slave is polled after transmission. Only when the slave reports a *ready status* does the SPI 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*. †

Related Information

[National Semiconductor Microwire Protocol](#) on page 19-19

SPI Slave

The SPI slave handles serial communication with transfer initiated and controlled by serial master peripheral devices.

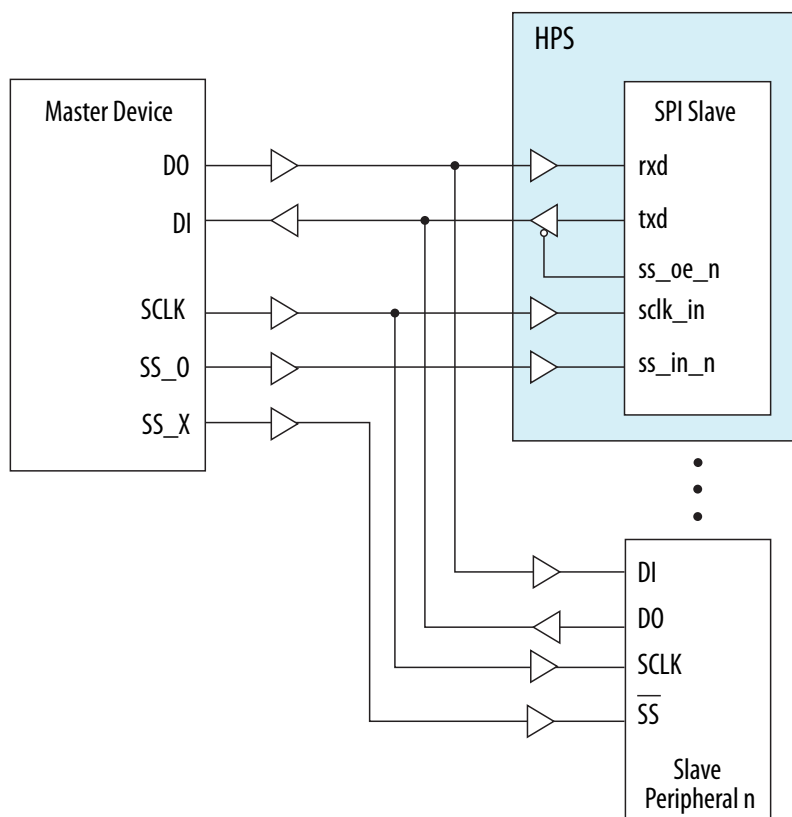
- `sclk_in`—serial clock to the SPI slave †
- `ss_in_n`—slave select input to the SPI slave †
- `ss_oe_n`—output enable for the SPI master or slave †
- `txd`—transmit data line for the SPI master or slave †
- `rx_d`—receive data line for the SPI master or slave †

When the SPI serial slave is selected, 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 SPI master device. Data are propagated from the serial slave on one edge of the serial clock line and sampled on the opposite edge. †

When the SPI 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 SPI master `rx_d` line. The buffers shown in the [Figure 19-5](#) figure are external to the SPI controller. `spi_oe_n` is the SPI slave output enable signal. †

The serial clock that regulates the data transfer is generated by the serial-master device and input to the SPI 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 SPI slave output enable (`ss_oe_n`) signal 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 SPI control register 0 (`CTRLR0`) can be programmed to inform the slave if it should respond with data from its `txd` line. †

Figure 19-5: SPI Slave



The `slv_oe` bit in the control register is only valid if the SPI slave interface is routed to the FPGA. To use the SPI slave in a multi master system or in a system that requires the SPI slave `TXD` to be tri-stated, you can do the following:

- If you want the SPI slave to control the tri-state of `TXD`, it must be routed to the FPGA first and use the FPGA IO. HPS I/O can also be used via the Loan I/O interface (timing permitting).
- If you do not want to route to the FPGA, then software control of the `TXD` (tri-state) must be performed with the already included code to control via an HPS GPIO input. Please refer to the pin connection guidelines to find which GPIO pins correspond to which HPS SPI slave `SS` ports.

Slave SPI and SSP Serial Transfers

“Motorola SPI Protocol” and the “Texas Instruments Synchronous Serial Protocol (SSP)” contain a description of the SPI and SSP serial protocols, respectively. †

If the SPI slave is *receive only* (`TMOD=2`), the transmit FIFO buffer 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=2. You should mask the Transmit FIFO Empty Interrupt when this mode is used. †

If the SPI slave transmits data to the master, you must ensure that data exists in the transmit FIFO buffer before a transfer is initiated by the serial-master device. If the master initiates a transfer to the SPI slave when no data exists in the transmit FIFO buffer, an error flag (TXE) is set in the SPI 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 (Transmit FIFO Empty Interrupt) the processor, indicating that the transmit FIFO buffer is nearly empty. When a DMA Controller is used, the DMA transmit data level register (DMATDLR) can be used to early request the DMA Controller, indicating that the transmit FIFO buffer is nearly empty. The FIFO buffer can then be refilled with data to continue the serial transfer. †

The receive FIFO buffer should be read each time the receive FIFO buffer 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 buffer is nearly full. When a DMA Controller is used, the DMA receive data level register (DMARDLR) can be used to early request the DMA controller, indicating that the receive FIFO buffer is nearly full. †

Related Information

- [Motorola SPI Protocol](#) on page 19-16
- [Texas Instruments Synchronous Serial Protocol \(SSP\)](#) on page 19-18

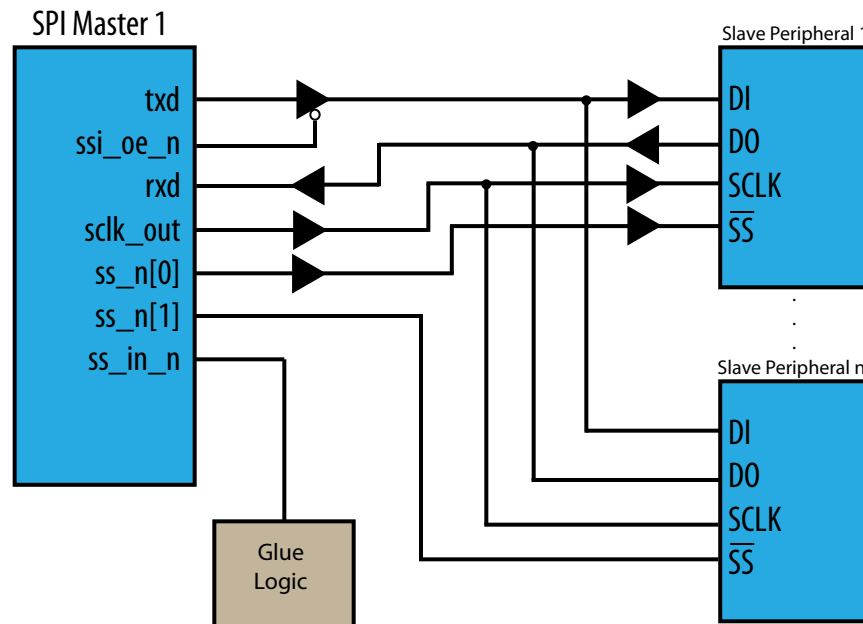
Serial Transfers

“National Semiconductor Microwire Protocol” describes the Microwire serial protocol in detail, including timing diagrams and information about how data are structured in the transmit and receive FIFO buffers before and after a serial transfer. The Microwire protocol operates in much the same way as the SPI protocol. There is no decode of the control frame by the SPI slave device. †

Related Information

[National Semiconductor Microwire Protocol](#) on page 19-19

Figure 19-7: SPI Configured as Master Device



Partner Connection Interfaces

The SPI can connect to any serial-master or serial-slave peripheral device using one of several interfaces.

Motorola SPI Protocol

With SPI, the clock polarity (SCPOL) configuration parameter determines whether the inactive state of the serial clock is high or low. The data frame can be 4 to 16 bits in length. †

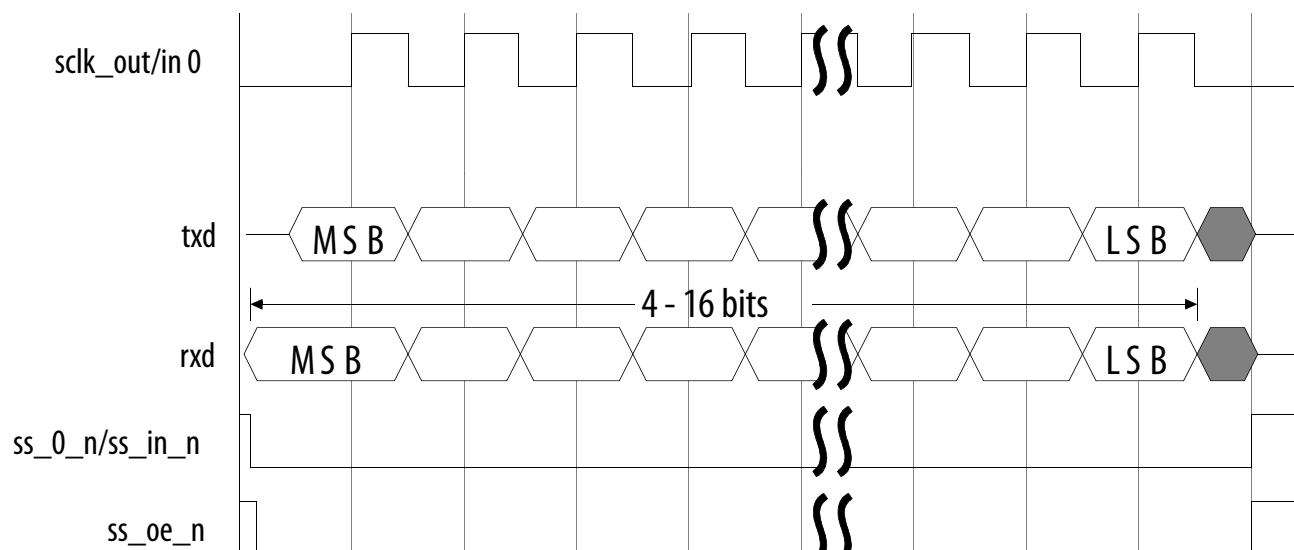
When the configuration parameter (SCPH = 0), data transmission begins on the falling edge of the slave select signal. The first data bit is captured by the master and slave peripherals on the first edge of the serial clock; therefore, valid data must be present on the `txd` and `rxn` lines prior to the first serial clock edge. †

The slave select signal takes effect only when used as slave SPI. For master SPI, the data transmission begins as soon as the output enable signal is deasserted.

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

- `sclk_out`—serial clock from SPI master †
- `sclk_in`—serial clock from SPI slave †
- `ss_0_n`—slave select signal from SPI master †
- `ss_oe_n`—output enable for the SPI master or slave †
- `txd`—transmit data line for the SPI master or slave †
- `rxn`—receive data line for the SPI master or slave †

Figure 19-8: SPI Serial Format (SCPH = 0)



There are four possible transfer modes on the SPI controller for performing SPI serial transactions; refer to “Transfer Modes”. For *transmit and receive transfers* (transfer mode field (9:8) of the Control Register 0 = 0), data transmitted from the SPI controller to the external serial device is written into the transmit FIFO buffer. Data received from the external serial device into the SPI controller is pushed into the receive FIFO buffer. †

Note: For *transmit only* transfers (transfer mode field (9:8) of the Control Register 0 = 1), data transmitted from the SPI controller to the external serial device is written into the transmit FIFO buffer. As the data received from the external serial device is deemed invalid, it is not stored in the SPI receive FIFO buffer. †

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

For EEPROM read transfers (transfer mode field [9:8] of the Control Register 0 = 3), the opcode or the EEPROM address, or both, are written into the transmit FIFO buffer. During transmission of these control frames, received data is not captured by the SPI master. After the control frames have been transmitted, receive data from the EEPROM is stored in the receive FIFO buffer.

SPI Serial Format

Two different modes of continuous data transfers are supported:

- When clock phase $SCPH = 0$ and clock polarity $SCPOL = 0$, the SPI Controller deasserts the slave select signal between each data word and the serial clock is held to its default value while the slave select signal is deasserted.†
- When $SCPH = 1$ and $SCPOL = 1$, the slave select is held asserted (active low) for the duration of the transfer.†

Figure 19-9: Serial Format Continuous Transfers ($SCPH = 0$ and $SCPOL = 0$)

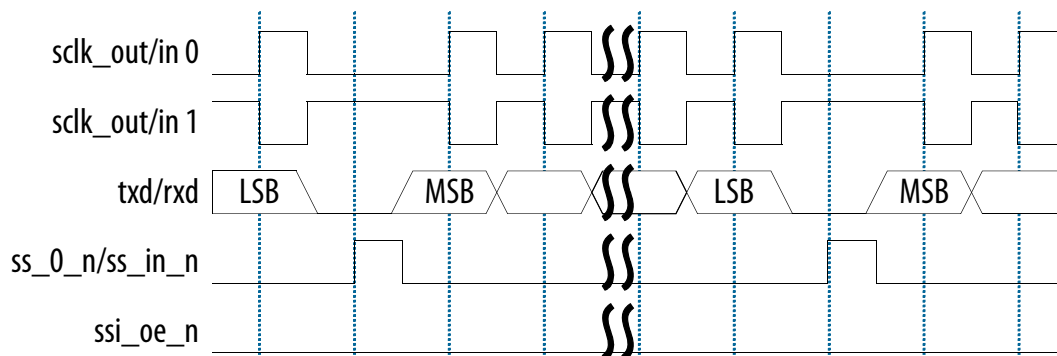
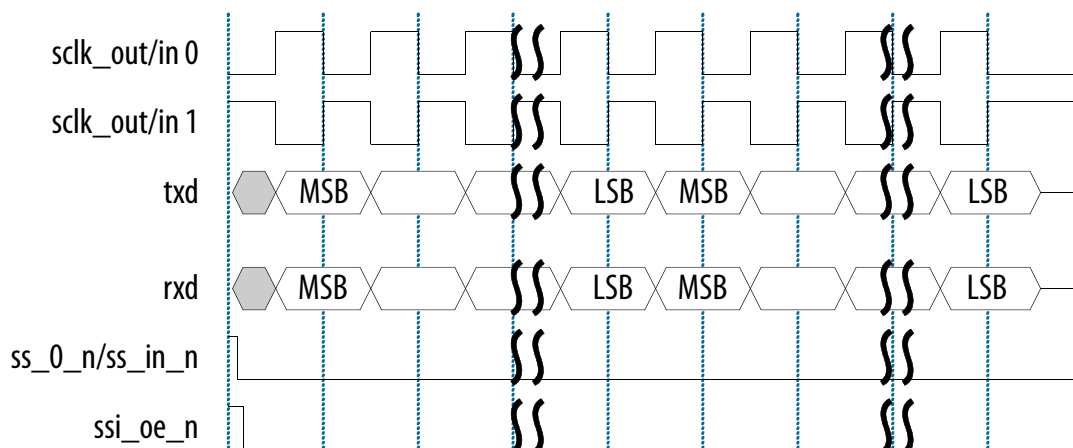


Figure 19-10: Serial Format Continuous Transfers ($SCPH = 1$ and $SCPOL = 1$)



Related Information

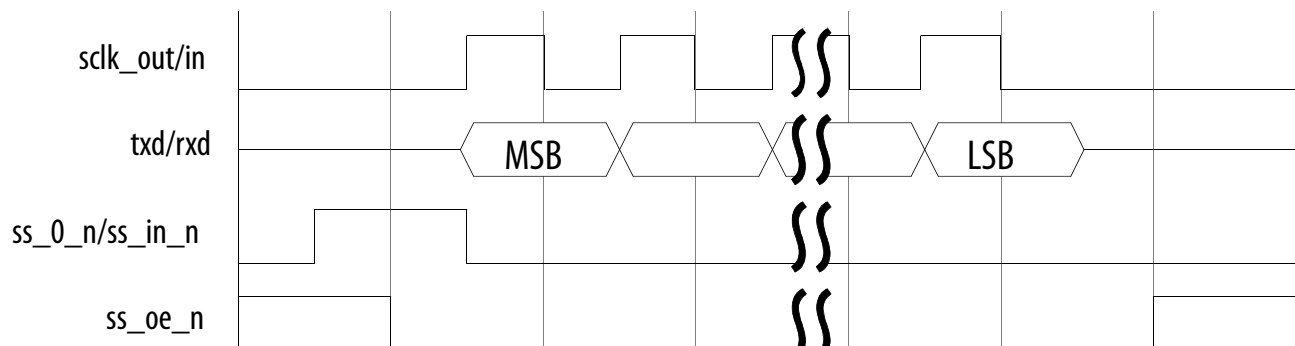
[Transfer Modes](#) on page 19-8

Texas Instruments Synchronous Serial Protocol (SSP)

Data transfers begin by asserting the frame indicator line (`ss_0_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 `rxn` 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.

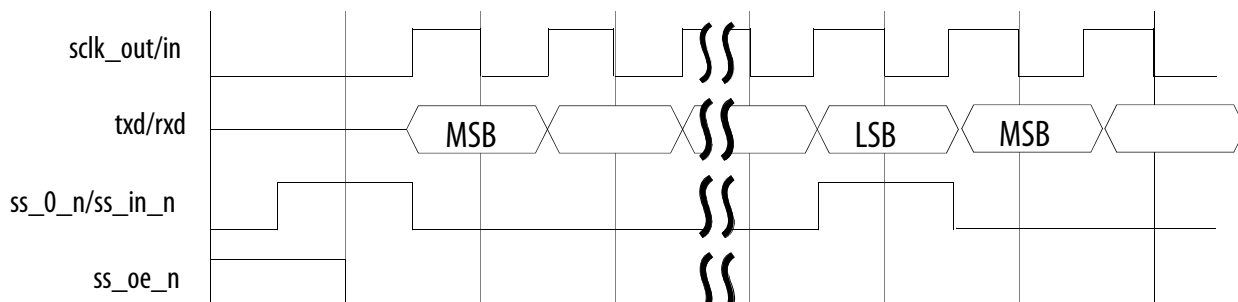
Note: The slave select signal (`ss_0_n`) takes effect only when used as slave SPI. For master SPI, the data transmission begins as soon as the output enable signal is deasserted.

Figure 19-11: 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 19-12: SSP Serial Format Continuous Transfer



National Semiconductor Microwire Protocol

For the master SPI, data transmission begins as soon as the output enable signal is deasserted. 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 SPI serial master. During this transmission, no data are present (high impedance) on the serial master's `rxn` 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 SPI 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. †

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

Nonsequential continuous transfers occur, with the control word for the next transfer following immediately after the LSB of the current data word. †

The only modification needed to perform a continuous nonsequential transfer is to write more control words into the transmit FIFO buffer. †

During sequential continuous transfers, only one control word is transmitted from the SPI 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 SPI master terminates the transfer when the number of words received is equal to the value in the CTRLR1 register plus one. †

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

Note: The SPI controller does not support continuous sequential Microwire writes, where $MDD = 1$ and $MWMOD = 1$. †

Continuous transfers occur with the control word for the next transfer following immediately after the LSB of the current data word.

The Microwire handshaking interface can also be enabled for SPI 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 SPI serial master checks for a ready status from the slave device before completing the transfer, or transmitting the next control word for continuous transfers. †

After the first data word has been transmitted to the serial-slave device, the SPI master polls the rxd input waiting for a ready status from the slave device. Upon reception of the ready status, the SPI master begins transmission of the next control word. After transmission of the last data frame has completed, the SPI master transmits a start bit to clear the ready status of the slave device before completing the transfer. †

In the SPI slave, data transmission begins with the falling edge of the slave select signal (ss_in_0). One-half serial clock ($sc1k\_in$) 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 $sc1k\_in$) by the SPI 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 SPI 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 SPI 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 deasserted one-half clock cycle later after the data are transferred. The SPI slave output enable signal is held inactive for the duration of the transfer. †

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

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

Figure 19-13: Single SPI Serial Master Microwire Serial Transfer (MDD=0)

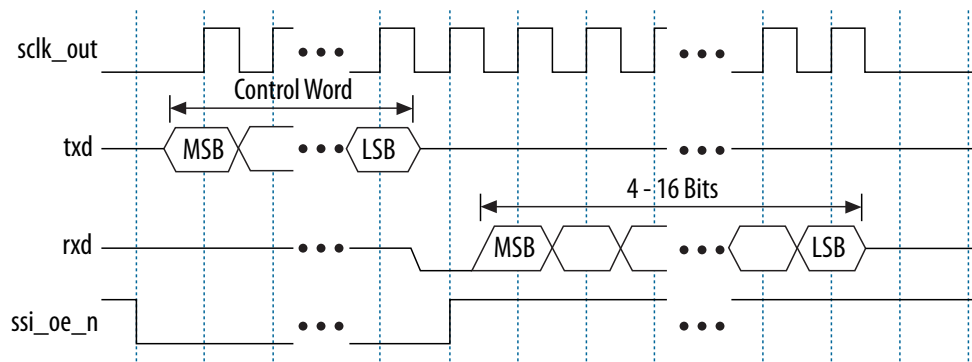
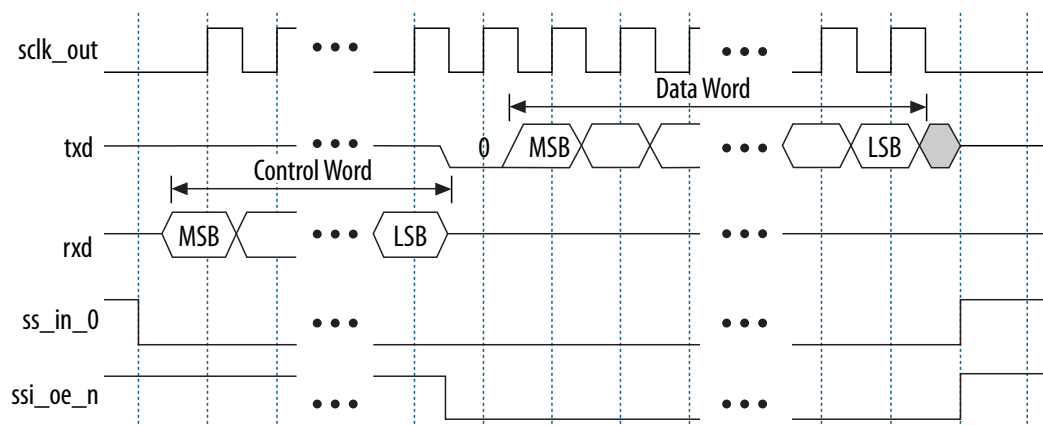


Figure 19-14: Single SPI Slave Microwire Serial Transfer (MDD=1)



DMA Controller Interface

The SPI controller supports DMA signaling to indicate when the receive FIFO buffer has data ready to be read or when the transmit FIFO buffer needs data. It requires two DMA channels, one for transmit data and one for receive data. The SPI controller can issue single or burst DMA transfers and accepts burst acknowledges from the DMA. System software can trigger the DMA burst mode by programming an appropriate value into the threshold registers. The typical setting of the threshold register value is half full.

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

Slave Interface

The host processor accesses data, control, and status information about the SPI controller through the slave interface. The SPI supports a data bus width of 32 bits.

Control and Status Register Access

Control and status registers within the SPI controller are byte-addressable. The maximum width of the control or status register in the SPI controller is 16 bits. Therefore, all read and write operations to the SPI control and status registers require only one access. †

Data Register Access

The data register (DR) within the SPI controller is 16 bits wide in order to remain consistent with the maximum serial transfer size (data frame). A write operation to DR moves data from the slave write data bus into the transmit FIFO buffer. An read operation from DR moves data from the receive FIFO buffer onto the slave readback data bus. †

Note: The DR register in the SPI controller occupies sixty-four 32-bit locations of the memory map to facilitate burst transfers. There are no burst transactions on the system bus itself, but SPI supports bursts on the system interconnect. Writing to any of these address locations has the same effect as pushing the data from the slave write data bus into the transmit FIFO buffer. Reading from any of these locations has the same effect as popping data from the receive FIFO buffer onto the slave readback data bus. The FIFO buffers on the SPI controller are not addressable.

Clocks and Resets

The SPI controller uses the clock and reset signals shown in the following table.

Table 19-6: SPI Controller Clocks and Resets

	Master	Slave
SPI clock	spi_m_clk	l4_main_clk
SPI bit-rate clock	sclk_out	sclk_in
Reset	spim_rst_n	spis_rst_n

Taking the SPI Controller Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

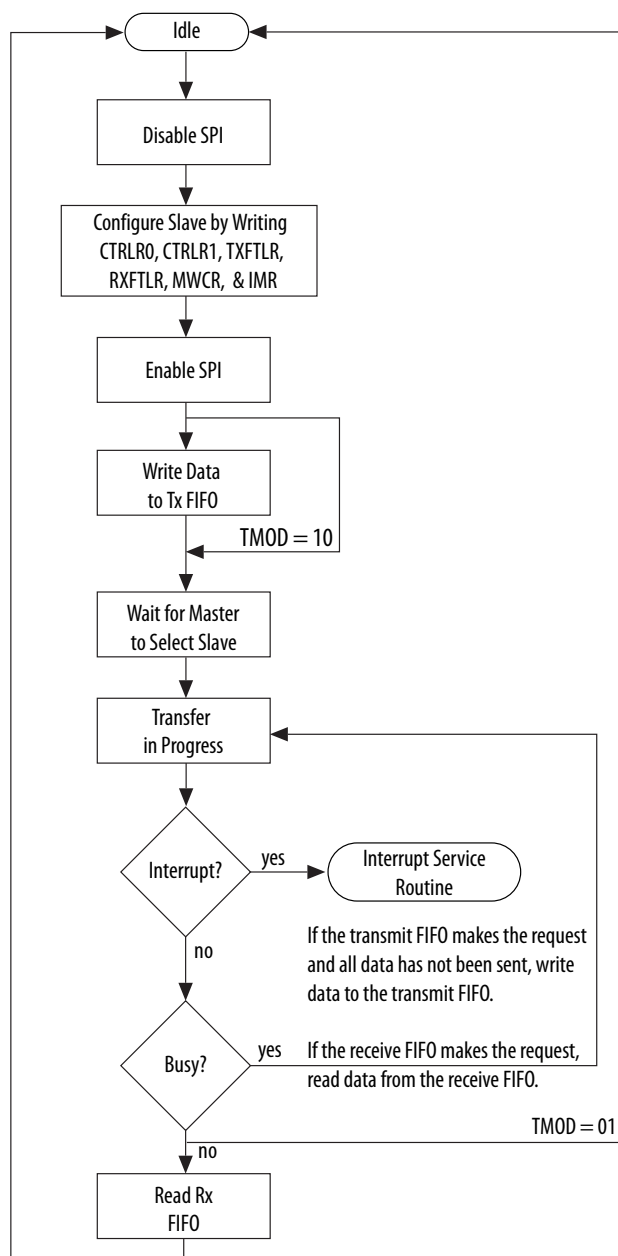
SPI Programming Model

This section describes the programming model for the SPI controller, for the following master and slave transfer types:

- Master SPI and SSP serial transfers
- Master Microwire serial transfers
- Slave SPI and SSP serial transfers
- Slave Microwire serial transfers
- Software Control for slave selection

Master SPI and SSP Serial Transfers

Figure 19-15: Master SPI or SSP Serial Transfer Software Flow



To complete an SPI or SSP serial transfer from the SPI master, follow these steps:

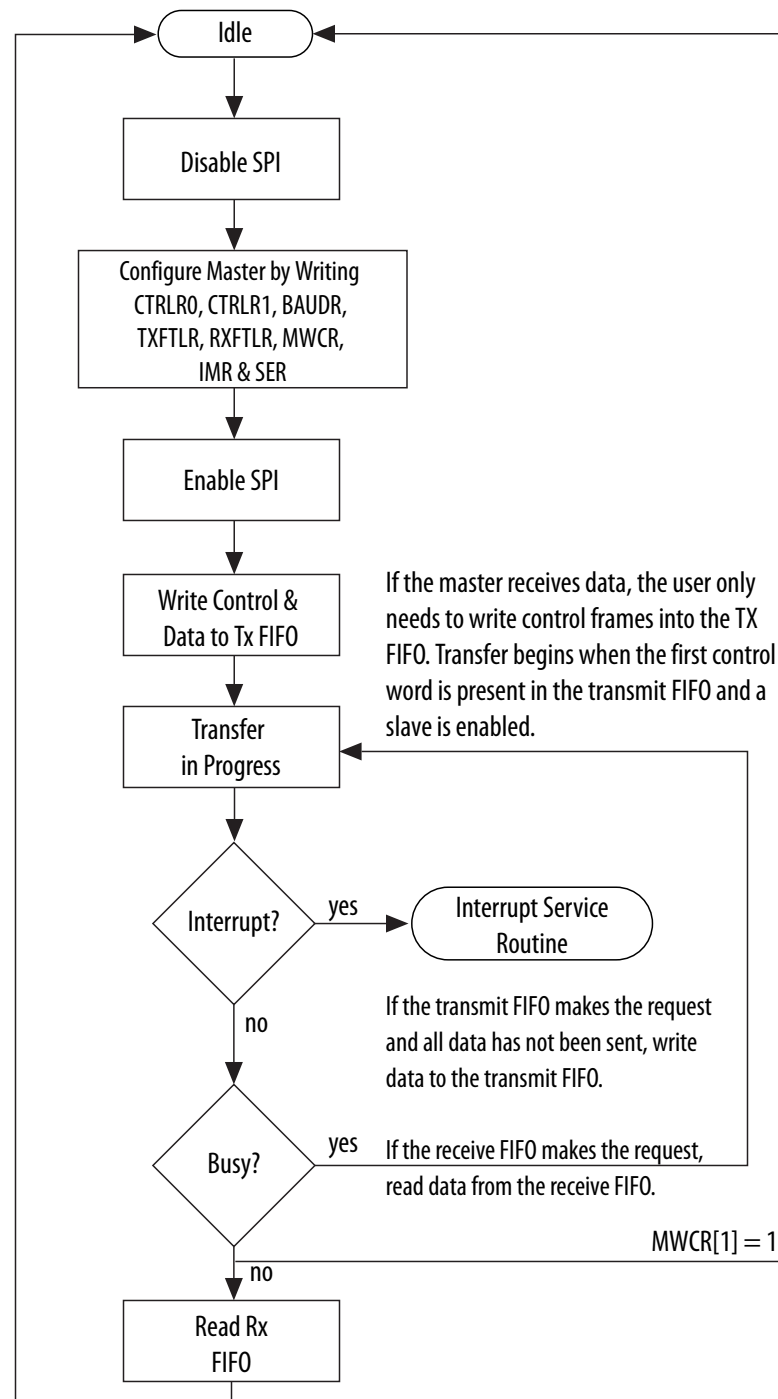
1. If the SPI master is enabled, disable it by writing 0 to the SSI Enable register (SSIENR).
2. Set up the SPI master control registers for the transfer. You can set these registers in any order.

- Write Control Register 0 (CTRLR0). For SPI transfers, you must set the serial clock polarity and serial clock phase parameters identical to the target slave device.
 - If the transfer mode is receive only, write Control Register 1 (CTRLR1) with the number of frames in the transfer minus 1. For example, if you want to receive four data frames, write 3 to this register.
 - 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) to set FIFO buffer threshold levels.
 - Write the IMR register to set up interrupt masks.
 - Write the Slave Enable Register (SER) register to enable the target slave for selection. If a slave is enabled at this time, the transfer begins as soon as one valid data entry is present in the transmit FIFO buffer. 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 SPI master by writing 1 to the SSIENR register.
 4. Write data for transmission to the target slave into the transmit FIFO buffer (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 the transfer to complete. If a transmit FIFO empty interrupt request is made, write the transmit FIFO buffer (write DR). If a receive FIFO full interrupt request is made, read the receive FIFO buffer (read DR).

6. The shift control logic stops the transfer when the transmit FIFO buffer is empty. If the transfer mode is receive only (TMOD = 2), the shift control logic stops the transfer 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 is not equal to 1), read the receive FIFO buffer until it is empty
8. Disable the SPI master by writing 0 to SSIENR.

Master Microwire Serial Transfers

Figure 19-16: Microwire Serial



To complete a Microwire serial transfer from the SPI master, follow these steps:

1. If the SPI master is enabled, disable it by writing 0 to SSIENR.
2. Set up the SPI control registers for the transfer. You can set these registers in any order.
 - Write CTRLR0 to set transfer parameters. If the transfer is sequential and the SPI master receives data, write CTRLR1 with the number of frames in the transfer minus 1. For example, if you want to receive four data frames, write 3 to this register.
 - Write BAUDR to set the baud rate for the transfer.
 - Write TXFTLR and RXFTLR to set FIFO buffer 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 buffer. 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 SPI master by writing 1 to the SSIENR register.
4. If the SPI master transmits data, write the control and data words into the transmit FIFO buffer (write DR). If the SPI master receives data, write the control word or words into the transmit FIFO buffer. 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 the transfer to complete. If a transmit FIFO empty interrupt request is made, write the transmit FIFO buffer (write DR). If a receive FIFO full interrupt request is made, read the receive FIFO buffer (read DR).
6. The shift control logic stops the transfer when the transmit FIFO buffer is empty. If the transfer mode is sequential and the SPI master receives data, the shift control logic stops the transfer when the specified number of data frames is received. When the transfer is done, the BUSY status is reset to 0.
7. If the SPI master receives data, read the receive FIFO buffer until it is empty.
8. Disable the SPI master by writing 0 to SSIENR.

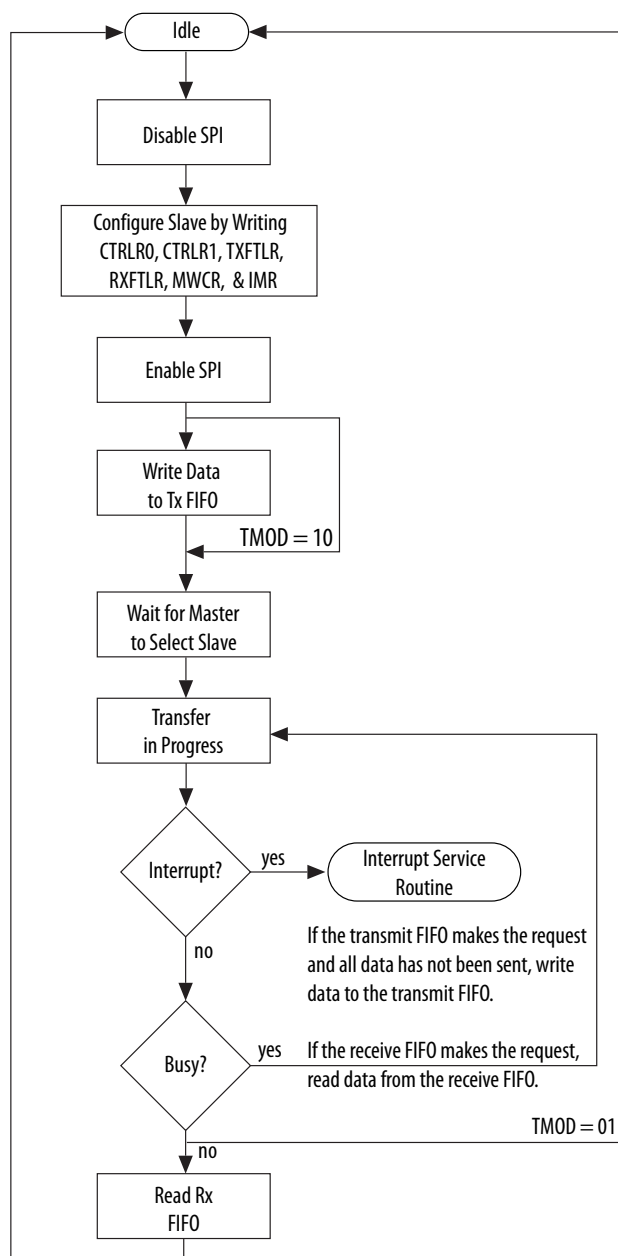
Related Information

[SPI Controller Address Map and Register Definitions](#) on page 19-33



Slave SPI and SSP Serial Transfers

Figure 19-17: Slave SPI or SSP Serial Transfer Software Flow



To complete a continuous serial transfer from a serial master to the SPI slave, follow these steps:

1. If the SPI slave is enabled, disable it by writing 0 to SSIENR.
2. Set up the SPI control registers for the transfer. You can set these registers in any order.

- Write CTRLR0 (for SPI transfers, set SCPH and SCPOL identical to the master device).
 - Write TXFTLR and RXFTLR to set FIFO buffer threshold levels.
 - Write the IMR register to set up interrupt masks.
3. Enable the SPI slave by writing 1 to the SSIENR register.
 4. If the transfer mode is transmit and receive (TMOD= 0) or transmit only (TMOD= 1), write data for transmission to the master into the transmit FIFO buffer (write DR). If the transfer mode is receive only (TMOD= 2), you need not write data into the transmit FIFO buffer. The current value in the transmit shift register is retransmitted.
 5. The SPI slave is now ready for the serial transfer. The transfer begins when a serial-master device selects the SPI slave.
 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 buffer (write DR). If a receive FIFO full interrupt request is made, read the receive FIFO buffer (read DR).
 7. The transfer ends when the serial master removes the select input to the SPI slave. When the transfer is completed, the BUSY status is reset to 0.
 8. If the transfer mode is not transmit only (TMOD != 1), read the receive FIFO buffer until empty.
 9. Disable the SPI slave by writing 0 to SSIENR.

Slave Microwire Serial Transfers

For the SPI slave, the Microwire protocol operates in much the same way as the SPI protocol. The SPI slave does not decode the control frame.

Software Control for Slave Selection

When using software to select slave devices, the input select lines from serial slave devices is connected to a single slave select output on the SPI master.

Example: Slave Selection Software Flow for SPI Master

1. If the SPI master is enabled, disable it by writing 0 to SSIENR.
2. Write CTRLR0 to match the required transfer.
3. If the transfer is receive only, write the number of frames into CTRLR1.
4. Write BAUDR to set the transfer baud rate.
5. Write TXFTLR and RXFTLR to set FIFO buffer threshold levels.
6. Write IMR register to set interrupt masks.
7. Write SER register bit 0 to 1 to select slave 1 in this example.
8. Write SSIENR register bit 0 to 1 to enable SPI master.

Example: Slave Selection Software Flow for SPI Slave

1. If the SPI slave is enabled, disable it by writing 0 to SSIENR.
2. Write CTRLR0 to match the required transfer.
3. Write TXFTLR and RXFTLR to set FIFO buffer threshold levels.
4. Write IMR register to set interrupt masks.
5. Write SSIENR register bit 0 to 1 to enable SPI slave.
6. If the SPI slave transmits data, write data into TX FIFO buffer.

Note: All other SPI slaves are disabled (SSIENR = 0) and therefore does not respond to an active level on their ss_in_n port.

The FIFO buffer depth (FIFO_DEPTH) for both the RX and TX buffers in the SPI controller is 256 entries.

DMA Controller Operation

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

Related Information

[DMA Controller](#) on page 16-1

For details about the DMA controller, refer to the *DMA Controller* chapter.

Transmit FIFO Buffer Underflow

During SPI serial transfers, transmit FIFO buffer requests are made to the DMA Controller whenever the number of entries in the transmit FIFO buffer is less or equal to the value in DMA Transmit Data Level Register (DMATDLR); also known as the watermark level. The DMA Controller responds by writing a burst of data to the transmit FIFO buffer, of length specified as DMA burst length.†

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

Related Information

[DMA Controller](#) on page 16-1

For details about the DMA burst length microcode setup, refer to the *DMA Controller* chapter.

Transmit FIFO Watermark

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 SPI transmits data to the rate at which the DMA can respond to destination burst requests. †

Example 1: Transmit FIFO Watermark Level = 64

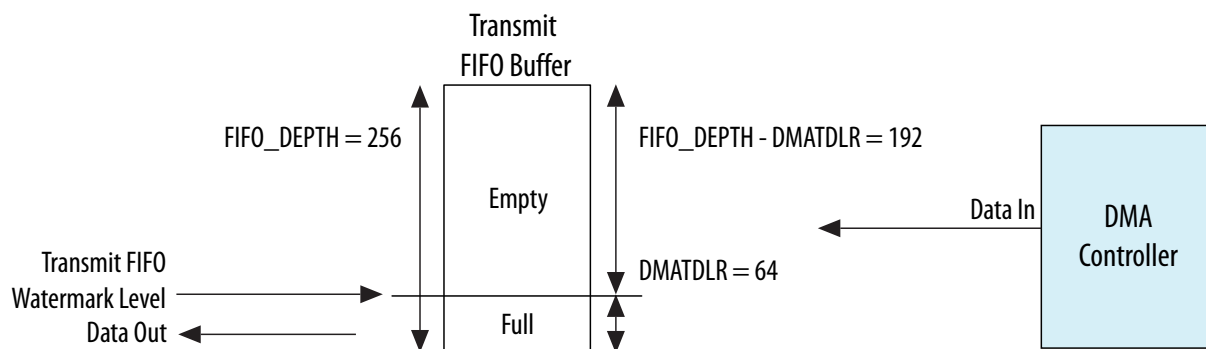
Consider the example where the assumption is made: †

$\text{DMA burst length} = \text{FIFO\_DEPTH} - \text{DMATDLR}$

Here the number of data items to be transferred in a DMA burst is equal to the empty space in the transmit FIFO buffer.

Consider the following:

- Transmit FIFO watermark level = DMATDLR = 64 †
- DMA burst length = FIFO_DEPTH - DMATDLR = 192 †
- SPI transmit FIFO_DEPTH = 256 †
- Block transaction size = 960 †

Figure 19-18: Transmit FIFO Watermark Level = 64

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

$$\text{Block transaction size/DMA burst length} = 960/192 = 5$$

The number of burst transactions in the DMA block transfer is 5. But the watermark level, DMATDLR, is quite low. Therefore, there is a high probability that the SPI serial transmit line needs to transmit data when there is no data left in the transmit FIFO buffer. This is a transmit underflow condition. This occurs because the DMA has not had time to service the DMA request before the FIFO buffer becomes empty.

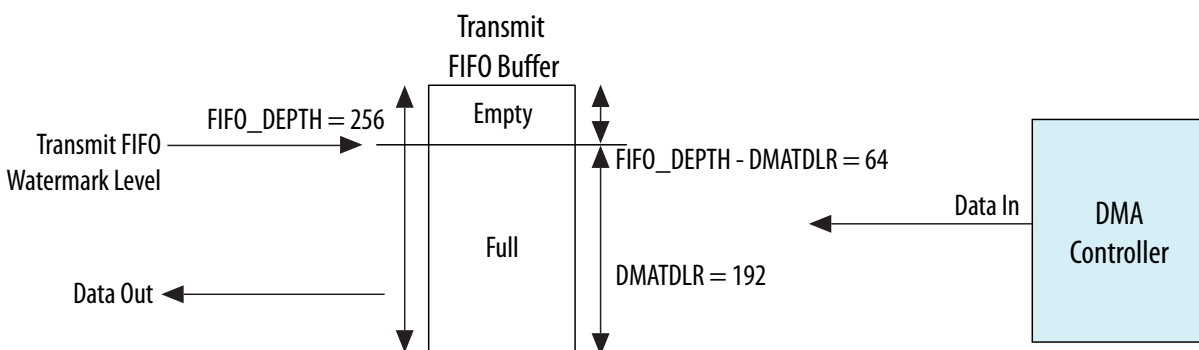
Example 2: Transmit FIFO Watermark Level = 192

Consider the example where the assumption is made: †

$$\text{DMA burst length} = \text{FIFO\_DEPTH} - \text{DMATDLR}$$

Here the number of data items to be transferred in a DMA burst is equal to the empty space in the transmit FIFO buffer. Consider the following:

- Transmit FIFO watermark level = DMATDLR = 192 †
- DMA burst length = FIFO_DEPTH - DMATDLR = 64 †
- SPI transmit FIFO_DEPTH = 256 †
- Block transaction size = 960 †

Figure 19-19: Transmit FIFO Watermark Level = 192

Number of burst transactions in block: †

Block transaction size/DMA burst length = $960/64 = 15$ †

In this block transfer, there are 15 destination burst transactions in a DMA block transfer. But the watermark level, DMATDLR, is high. Therefore, the probability of SPI transmit underflow is low because the DMA controller has plenty of time to service the destination burst transaction request before the SPI transmit FIFO buffer becomes empty. †

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

Transmit FIFO Buffer Overflow

Setting the DMA transaction burst length to a value greater than the watermark level that triggers the DMA request may cause overflow when there is not enough space in the transmit FIFO buffer to service the destination burst request. Therefore, the following equation must be adhered to in order to avoid overflow: †

$$\text{DMA burst length} \leq \text{FIFO\_DEPTH} - \text{DMATDLR}$$

In Example 2: Transmit Watermark Level = 192, the amount of space in the transmit FIFO buffer at the time of the burst request is made is equal to the DMA burst length. Thus, the transmit FIFO buffer may be full, but not overflowed, at the completion of the burst transaction.

Therefore, for optimal operation, DMA burst length should be set at the FIFO buffer level that triggers a transmit DMA request; that is: †

$$\text{DMA burst length} = \text{FIFO\_DEPTH} - \text{DMATDLR}$$

Adhering to this equation reduces the number of DMA bursts needed for block transfer, and this in turn improves bus utilization. †

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

Related Information

[Transmit FIFO Watermark](#) on page 19-30

Receive FIFO Buffer Overflow

During SPI serial transfers, receive FIFO buffer requests are made to the DMA whenever the number of entries in the receive FIFO buffer is at or above the DMA Receive Data Level Register, that is DMARDLR + 1. This is known as the watermark level. The DMA responds by fetching a burst of data from the receive FIFO buffer. †

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

Choosing Receive Watermark Level

Similar to choosing the transmit watermark level, the receive watermark level, DMARDLR + 1, should be set to minimize the probability of overflow. It is a trade off between the number of DMA burst transactions required per block versus the probability of an overflow occurring. †

Related Information

[Texas Instruments Synchronous Serial Protocol \(SSP\)](#) on page 19-18

Receive FIFO Buffer Underflow

Setting the source transaction burst length greater than the watermark level can cause underflow where there is not enough data to service the source burst request. Therefore, the following equation must be adhered to avoid underflow: †

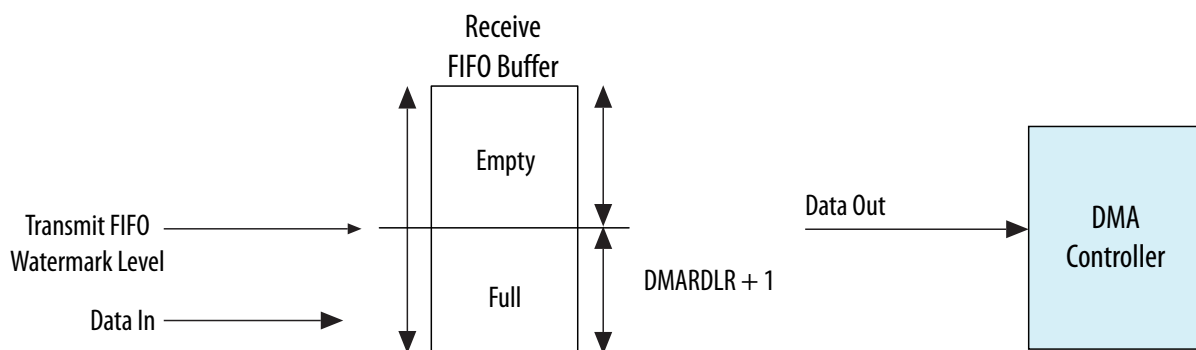
$$\text{DMA burst length} = \text{DMATDLR} + 1$$

If the number of data items in the receive FIFO buffer is equal to the source burst length at the time of the burst request is made, the receive FIFO buffer may be emptied, but not underflowed, at the completion of the burst transaction. For optimal operation, DMA burst length should be set at the watermark level, $\text{DMATDLR} + 1$. †

Adhering to this equation reduces the number of DMA bursts in a block transfer, which in turn can improve bus utilization. †

Note: The receive FIFO buffer is not be empty at the end of the source burst transaction if the SPI controller has successfully received one data item or more on the serial receive line during the burst. †

Figure 19-20: Receive FIFO Buffer



SPI Controller Address Map and Register Definitions

The address map and register definitions for the HPS-FPGA bridges consist of the following regions:

- SPI Slave Module 0
- SPI Slave Module 1
- SPI Master Module 0
- SPI Master Module 1

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter.
- <https://www.intel.com/content/www/us/en/programmable/hps/arria-v/hps.html>

2021.05.07

av_5v4



Subscribe



Send Feedback

The I²C controller provides support for a communication link between integrated circuits on a board. It is a simple two-wire bus which consists of a serial data line (SDA) and a serial clock (SCL) for use in applications such as temperature sensors and voltage level translators to EEPROMs, A/D and D/A converters, CODECs, and many types of microprocessors. †

The hard processor system (HPS) provides four I²C controllers to enable system software to communicate serially with I²C buses. Each I²C controller can operate in master or slave mode, and support standard mode of up to 100 Kbps or fast mode of up to 400 Kbps. These I²C controllers are instances of the Synopsys DesignWare APB I²C (DW_apb_i2c) controller.

Each I²C controller must be programmed to operate in either master or slave mode only. Operating as a master and slave simultaneously is not supported. † ⁽⁵⁷⁾

Features of the I²C Controller

The I²C controller has the following features:

- Maximum clock speed of up to 400 Kbps
- One of the following I²C operations:
 - A master in an I²C system and programmed only as a master †
 - A slave in an I²C system and programmed only as a slave †
- 7- or 10-bit addressing †
- Mixed read and write combined-format transactions in both 7-bit and 10-bit addressing mode †
- Bulk transmit mode †
- Transmit and receive buffers †
- Handles bit and byte waiting at all bus speeds †
- DMA handshaking interface †

⁽⁵⁷⁾ Portions © 2017 Synopsys, Inc. Used with permission. All rights reserved. Synopsys & DesignWare are registered trademarks of Synopsys, Inc. All documentation is provided "as is" and without any warranty. Synopsys expressly disclaims any and all warranties, express, implied, or otherwise, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, and any warranties arising out of a course of dealing or usage of trade.

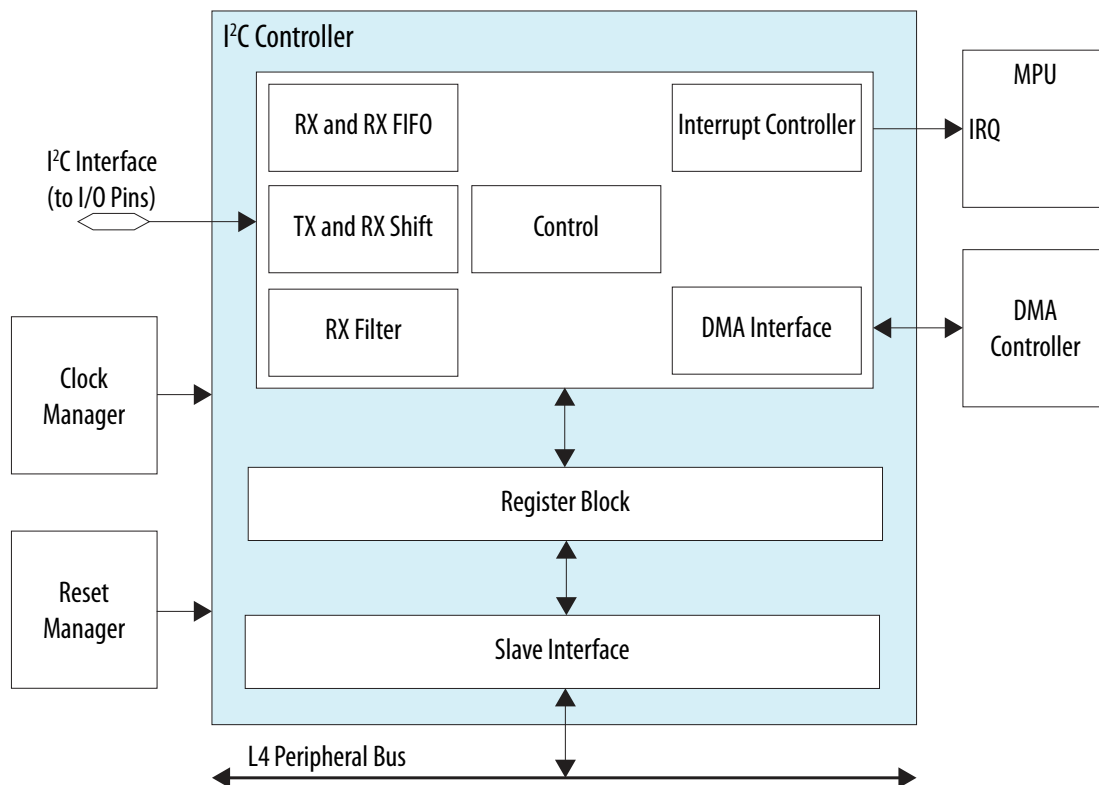
†Paragraphs marked with the dagger (†) symbol are Synopsys Proprietary. Used with permission.

I²C Controller Block Diagram and System Integration

The I²C controller consists of an internal slave interface, an I²C interface, and FIFO logic to buffer data between the two interfaces. †

The host processor accesses data, control, and status information about the I²C controller through a 32-bit slave interface.

Figure 20-1: I²C Controller Block Diagram



The I²C controller consists of the following modules and interfaces:

- Slave interface for control and status register (CSR) accesses and DMA transfers, allowing a master to access the CSRs and the DMA to read or write data directly.
- Two FIFO buffers for transmit and receive data, which hold the Rx FIFO and Tx FIFO buffer register banks and controllers, along with their status levels. †
- Shift logic for parallel-to-serial and serial-to-parallel conversion
 - Rx shift – Receives data into the design and extracts it in byte format. †
 - Tx shift – Presents data supplied by CPU for transfer on the I²C bus. †
- Control logic responsible for implementing the I²C protocol.

- DMA interface that generates handshaking signals to the DMA controller in order to automate the data transfer without CPU intervention. †
- Interrupt controller that generates raw interrupts and interrupt flags, allowing them to be set and cleared. †
- Receive filter for detecting events, such as start and stop conditions, in the bus; for example, start, stop and arbitration lost. †

I²C Controller Signal Description

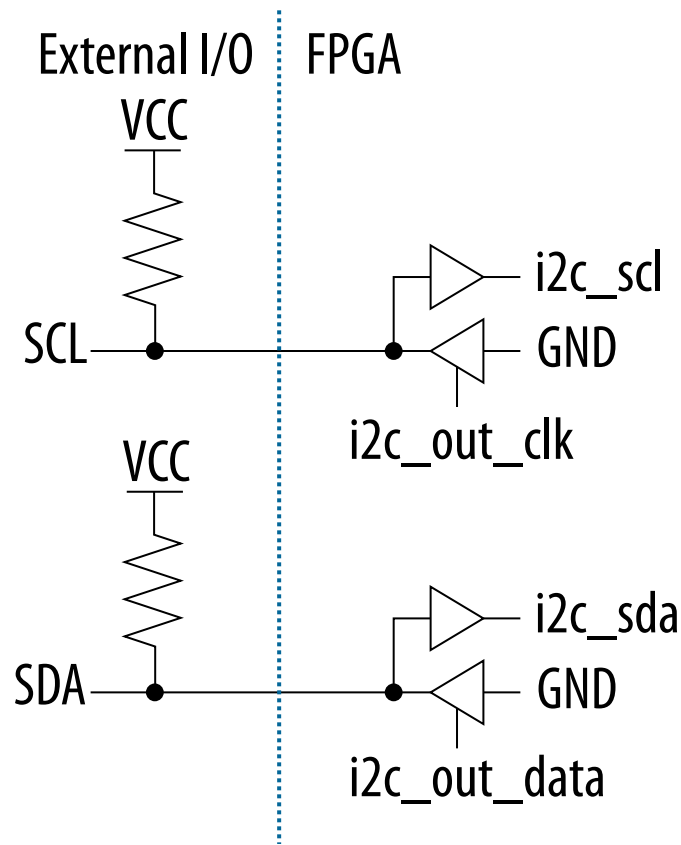
All instances of the I²C controller connect to external pins through pin multiplexers. Pin multiplexing allows all instances to function simultaneously and independently. The pins must be connected to a pull-up resistors and the I²C bus capacitance cannot exceed 400 pF.

Table 20-1: I²C Controller Interface HPS I/O Pins

Pin Name	Signal Width	Direction	Description
SCL	1 bit	Bidirectional	Serial clock
SDA	1 bit	Bidirectional	Serial data

Table 20-2: HPS I²C Signals for FPGA Routing

Signal Name	Signal Width	Direction	Description
i2c<#>_scl	1 bit	Input	Incoming I ² C clock source. This is the input SCL signal
i2c<#>_out_clk	1 bit	Output	Outgoing I ² C clock enable. Output SCL signal. This signal is logically inverted and is synchronous to the HPS peripheral clock
i2c<#>_sda	1 bit	Input	Incoming I ² C data. This is the input SDA signal.
i2c<#>_out_data	1 bit	Output	Outgoing I ² C data enable. Output SDA signal. This signal is logically inverted and is synchronous to the HPS peripheral clock.

Figure 20-2: I²C Interface in FPGA Fabric

The figure above shows the typical connection on the I²C interface in FPGA fabric with `alt_iobuff`.

For both I²C clock and data, external IO pins are open drain connection. When output enables `i2c_out_data` and `i2c_out_clk` are asserted, external signal will be driven to ground.

Related Information

[I/O Buffer \(ALTIOBUF\)](#)

For more information on configuring open drain I/O buffer to connect I²C signals to external I/O pins, please refer to [Altera I/O Buffer \(ALTIOBUF\) Megafunction User Guide](#).

Functional Description of the I²C Controller

Feature Usage

The I²C controller can operate in standard mode (with data rates of up to 100 Kbps) or fast mode (with data rates less than or equal to 400 Kbps). Additionally, fast mode devices are downward compatible. For instance, fast mode devices can communicate with standard mode devices in 0 to 100 Kbps I²C bus system. However, standard mode devices are not upward compatible and should not be incorporated in a fast-mode I²C bus system as they cannot follow the higher transfer rate and therefore unpredictable states would occur. †

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

Behavior

You can control the I²C controller via software to be in either mode:

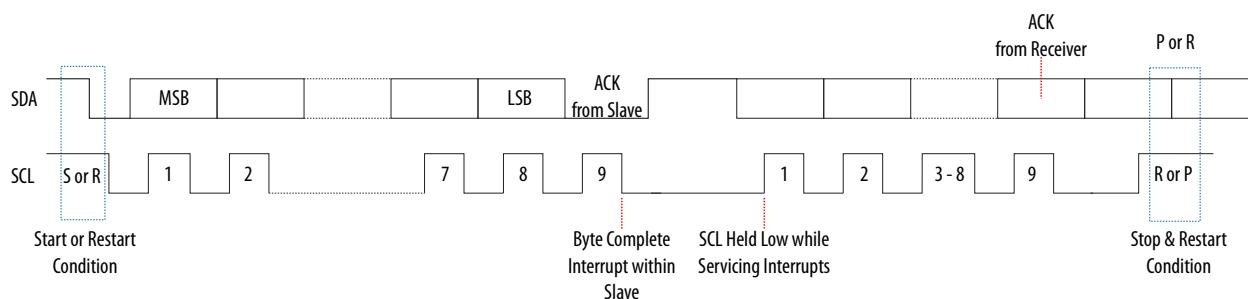
- An I²C master only, communicating with other I²C slaves.
- An I²C slave only, communicating with one or more I²C masters.

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

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

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

Figure 20-3: Data Transfer on the I²C Bus †



The I²C controller is a synchronous serial interface. The SDA line is a bidirectional signal and changes only while the SCL line is low, except for STOP, START, and RESTART conditions. The output drivers are open-drain or open-collector to perform wire-AND functions on the bus. The maximum number of devices on the bus is limited by only the maximum capacitance specification of 400 pF. Data is transmitted in byte packages. †

START and STOP Generation

When operating as a master, putting data into the transmit FIFO causes the I²C controller to generate a START condition on the I²C bus. In order for the master to complete the transfer and issue a STOP condition it must find a transmit FIFO entry tagged with a stop bit. Allowing the transmit FIFO to empty without a stop bit set, the master will stall the transfer by holding the SCL line low. †

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

Combined Formats

The I²C controller supports mixed read and write combined format transactions in both 7-bit and 10-bit addressing modes. †

The I²C controller does not support mixed address and mixed address format—that is, a 7-bit address transaction followed by a 10-bit address transaction or vice versa—combined format transactions. †

To initiate combined format transfers, the IC_RESTART_EN bit in the IC_CON register should be set to 1. With this value set and operating as a master, when the I²C controller completes an I²C transfer, it checks the transmit FIFO and executes the next transfer. If the direction of this transfer differs from the previous transfer, the combined format is used to issue the transfer. If the IC_RESTART_EN is 0, a STOP is issued followed by a START condition. Another way to generate the RESTART condition is to set the Restart bit [10] of the DATA_CMD register. Regardless if the direction of the transfer changes or not the RESTART condition is generated. †

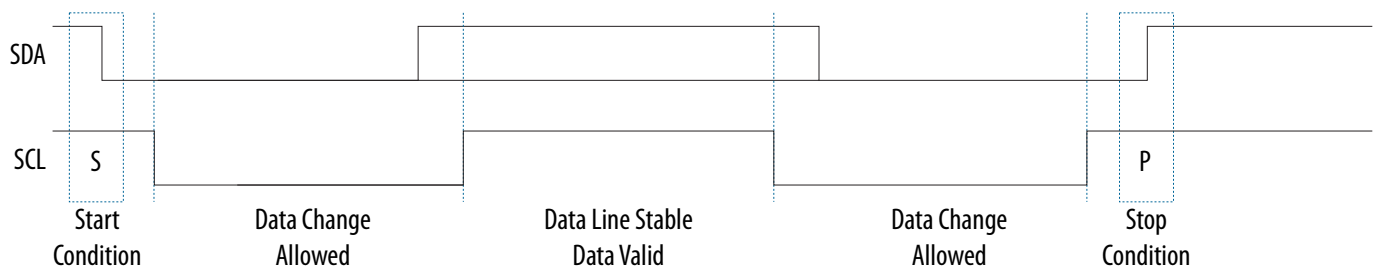
Protocol Details

START and STOP Conditions

When the bus is idle, both the SCL and SDA signals are pulled high through pull-up resistors on the bus. When the master wants to start a transmission on the bus, the master issues a START condition. This is defined to be a high-to-low transition of the SDA signal while SCL is 1. When the master wants to terminate the transmission, the master issues a STOP condition. This is defined to be a low-to-high transition of the SDA line while SCL is 1. †

The following figure shows the timing of the START and STOP conditions. When data is being transmitted on the bus, the SDA line must be stable when SCL is 1. †

Figure 20-4: Timing Diagram for START and STOP Conditions



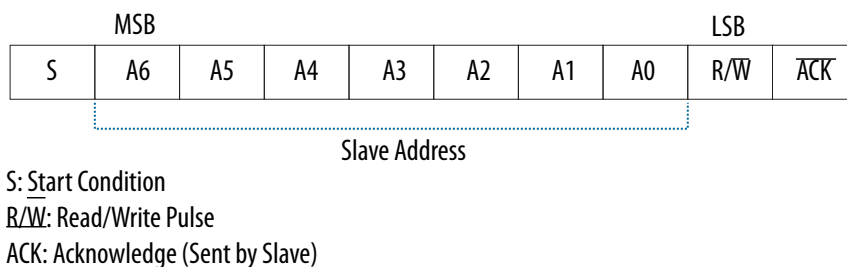
The signal transitions for the START or STOP condition, as shown in the figure, reflect those observed at the output signals of the master driving the I²C bus. Care should be taken when observing the SDA or SCL signals at the input signals of the slave(s), because unequal line delays may result in an incorrect SDA or SCL timing relationship. †

Addressing Slave Protocol

7-Bit Address Format

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

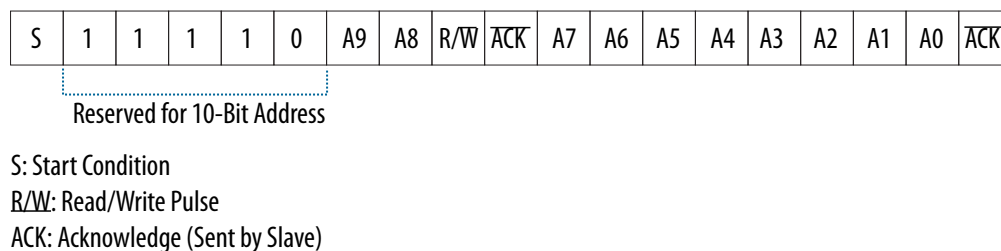
Figure 20-5: 7- Bit Address Format



10-Bit Address Format

During 10-bit addressing, two bytes are transferred to set the 10-bit address. The transfer of the first byte contains the following bit definition. The first five bits (bits 7:3) notify the slaves that this is a 10-bit transfer followed by the next two bits (bits 2:1), which set the slaves address bits 9:8, and the LSB bit (bit 0) is the R/W bit. The second byte transferred sets bits 7:0 of the slave address. †

Figure 20-6: 10-Bit Address Format



The following table defines the special purpose and reserved first byte addresses. †

Table 20-3: I²C Definition of Bits in First Byte

Slave Address	R/W Bit	Description
0000 000	0	General call address. The I ² C controller places the data in the receive buffer and issues a general call interrupt.
0000 000	1	START byte. For more details, refer to “START BYTE Transfer Protocol”
0000 001	X	CBUS address. The I ² C controller ignores these accesses.
0000 010	X	Reserved
0000 011	X	Reserved
0000 1XX	X	Unused
1111 1XX	X	Reserved
1111 0XX	X	10-bit slave addressing.

Note to Table: ‘X’ indicates do not care.

Related Information

[START BYTE Transfer Protocol](#) on page 20-10

Transmitting and Receiving Protocol

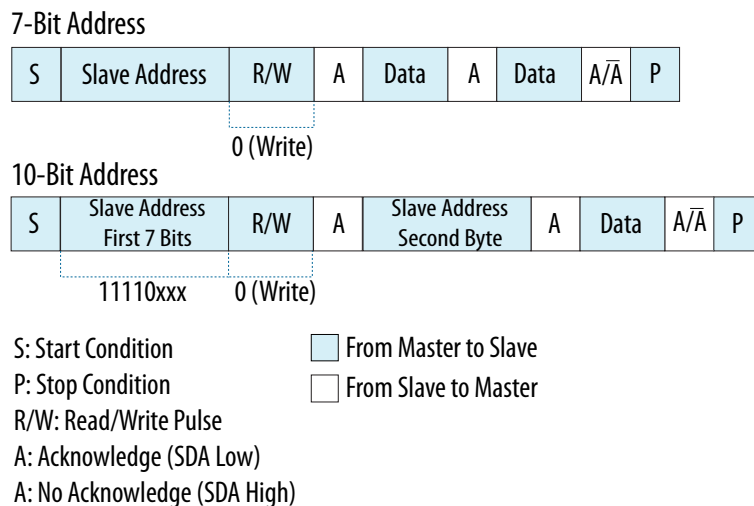
The master can initiate data transmission and reception to or from the bus, acting as either a master-transmitter or master-receiver. A slave responds to requests from the master to either transmit data or receive data to or from the bus, acting as either a slave-transmitter or slave-receiver, respectively.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

Master-Transmitter and Slave-Receiver

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

If the master-transmitter is transmitting data as shown in the following figure, then the slave-receiver responds to the master-transmitter with an ACK pulse after every byte of data is received. †

Figure 20-7: Master-Transmitter Protocol †



Master-Receiver and Slave-Transmitter

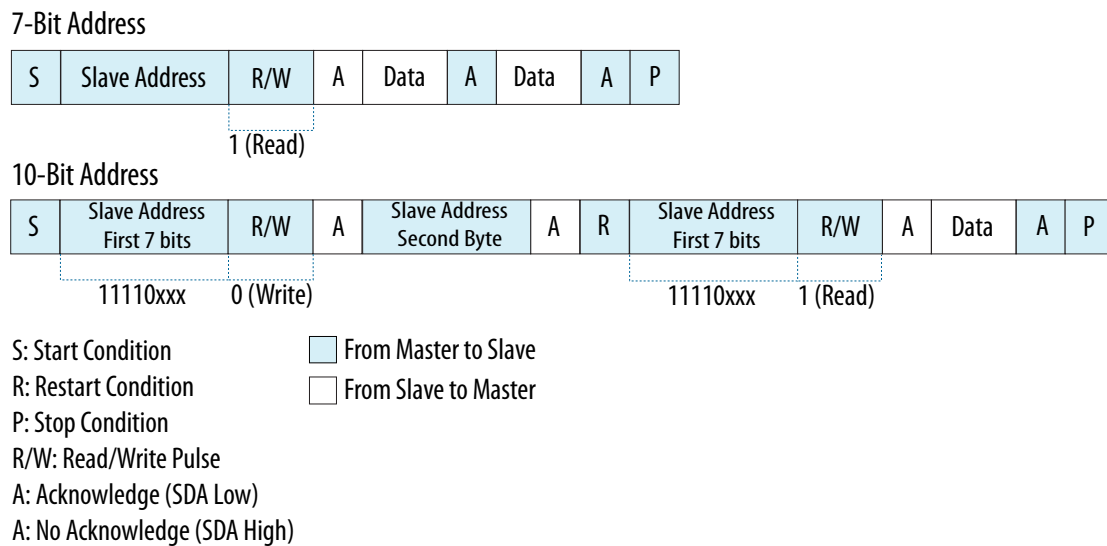
If the master is receiving data as shown in the following figure, then the master responds to the slave-transmitter with an ACK pulse after a byte of data has been received, except for the last byte. This is the way the master-receiver notifies the slave-transmitter that this is the last byte. The slave-transmitter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

When a master does not want to relinquish the bus with a STOP condition, the master can issue a RESTART condition. This is identical to a START condition except it occurs after the ACK pulse. Operating in master mode, the I²C controller can then communicate with the same slave using a transfer of a different direction. For a description of the combined format transactions that the I²C controller supports, refer to “Combined Formats” section of this chapter. †

Note: The I²C controller must be inactive on the serial port before the target slave address register, IC_TAR, can be reprogrammed. †

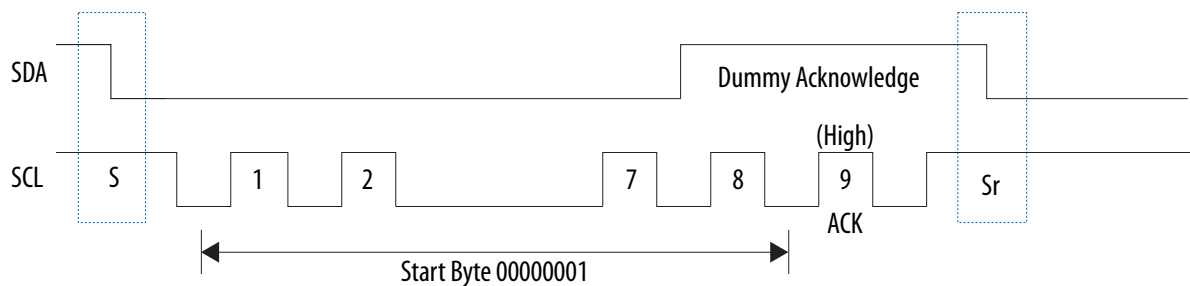


Figure 20-8: Master-Receiver Protocol †

**Related Information****Combined Formats** on page 20-6**START BYTE Transfer Protocol**

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

Figure 20-9: START BYTE Transfer †



The START BYTE has the following procedure: †

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

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

Multiple Master Arbitration

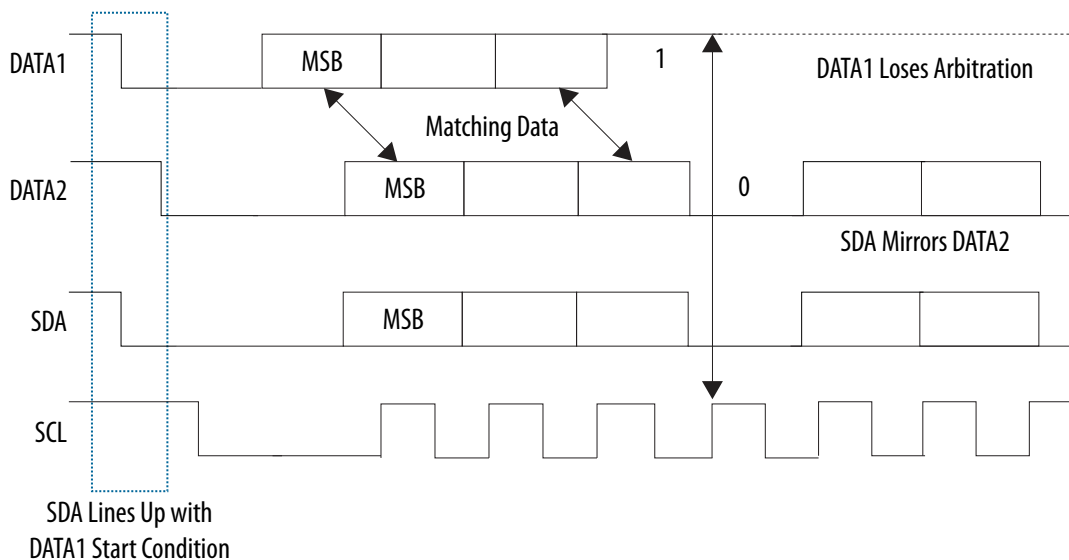
The I²C controller bus protocol allows multiple masters to reside on the same bus. If there are two masters on the same I²C-bus, there is an arbitration procedure if both try to take control of the bus at the same time by simultaneously generating a START condition. Once a master (for example, a microcontroller)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

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

Upon detecting that it has lost arbitration to another master, the I²C controller stops generating SCL. †

The following figure illustrates the timing of two masters arbitrating on the bus.

Figure 20-10: Multiple Master Arbitration †



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

Arbitration is not allowed between the following conditions: †

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

Slaves are not involved in the arbitration process. †

Clock Synchronization

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

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

Figure 20-11: Multiple Master Clock Synchronization †

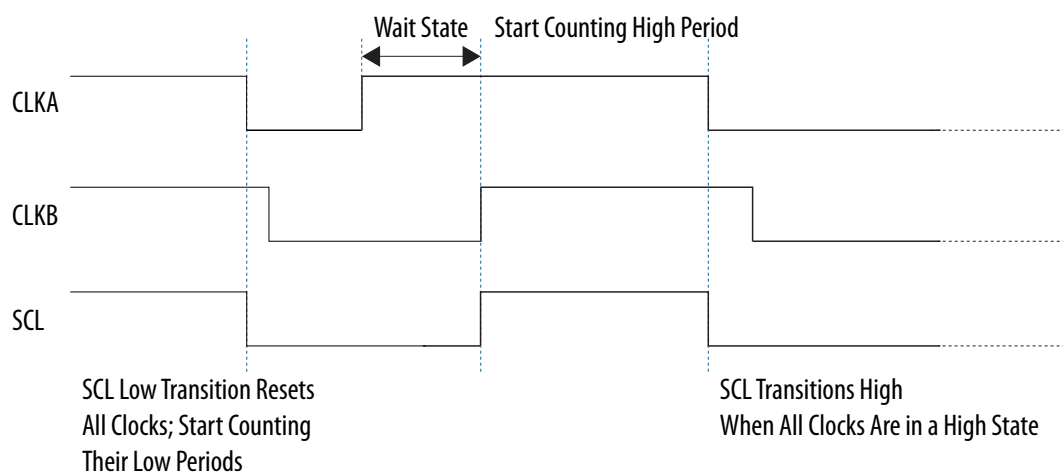
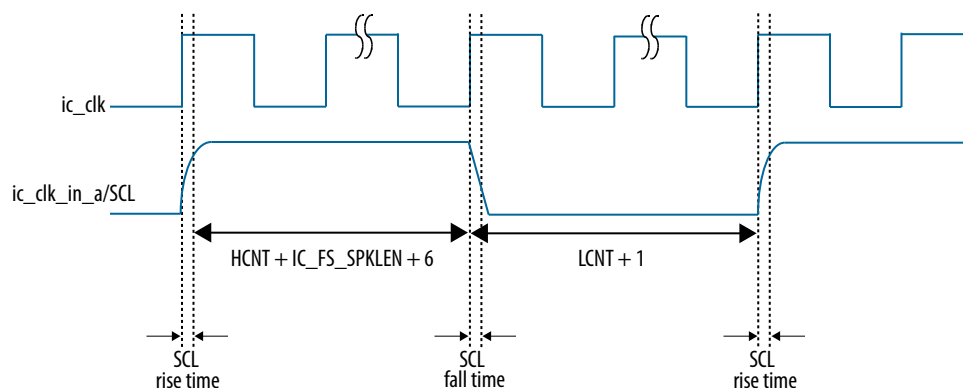


Figure 20-12: Impact of SCL Rise Time and Fall Time on Generated SCL

The following equations can be used to compute SCL high and low time:

$$SCL\_High\_time = [(HCNT + IC\_FS\_SPKLEN + 6) * ic\_clk] + SCL\_Fall\_time$$

$$SCL\_Low\_time = [(LCNT + 1) * ic\_clk] - SCL\_Fall\_time + SCL\_Rise\_time$$

Clock Frequency Configuration

When you configure the I²C controller as a master, the SCL count registers must be set before any I²C bus transaction can take place in order to ensure proper I/O timing. † There are four SCL count registers:

- Standard speed I²C clock SCL high count, `IC_SS_SCL_HCNT` †
- Standard speed I²C clock SCL low count, `IC_SS_SCL_LCNT` †
- Fast speed I²C clock SCL high count, `IC_FS_SCL_HCNT` †
- Fast speed I²C clock SCL low count, `IC_FS_SCL_LCNT` †

It is not necessary to program any of the SCL count registers if the I²C controller is enabled to operate only as an I²C slave, since these registers are used only to determine the SCL timing requirements for operation as an I²C master. †

Minimum High and Low Counts

When the I²C controller operates as an I²C master in both transmit and receive transfers, the minimum value that can be programmed in the SCL low count registers is 8 while the minimum value allowed for the SCL high count registers is 6. †

The minimum value of 8 for the low count registers is due to the time required for the I²C controller to drive SDA after a negative edge of SCL. The minimum value of 6 for the high count register is due to the time required for the I²C controller to sample SDA during the high period of SCL. †

The I²C controller adds one cycle to the low count register values in order to generate the low period of the SCL clock.

The I²C controller adds seven cycles to the high count register values in order to generate the high period of the SCL clock. This is due to the following factors: †



- The digital filtering applied to the SCL line incurs a delay of four $\text{t}_{4\_sp\_clk}$ cycles. This filtering includes metastability removal and a 2-out-of-3 majority vote processing on SDA and SCL edges. †
- Whenever SCL is driven 1 to 0 by the I²C controller—that is, completing the SCL high time—an internal logic latency of three $\text{t}_{4\_sp\_clk}$ cycles incurs. †

Consequently, the minimum SCL low time of which the I²C controller is capable is nine (9) $\text{t}_{4\_sp\_clk}$ periods (8+1), while the minimum SCL high time is thirteen (13) $\text{t}_{4\_sp\_clk}$ periods (6+1+3+3). †

Note: The `ic_fs_spklen` register must be set before any I²C bus transaction can take place to ensure stable operation. This register sets the duration measured in `ic_clk` cycles, of the longest spike in the SCL or SDA lines that is filtered out by the spike suppression logic. †

Calculating High and Low Counts

The calculations below show an example of how to calculate SCL high and low counts for each speed mode in the I²C controller.

The equation to calculate the proper number of $\text{t}_{4\_sp\_clk}$ clock pulses required for setting the proper SCL clocks high and low times is as follows: †

Table 20-4: Equation

$IC\_HCNT = \text{ceil}(\text{MIN\_SCL\_HIGHTime} * OSCFREQ)$

$IC\_LCNT = \text{ceil}(\text{MIN\_SCL\_LOWtime} * OSCFREQ)$

MIN_SCL_HIGHTime = minimum high period

MIN_SCL_HIGHTime =

4000 ns for 100 kbps

600 ns for 400 kbps

60 ns for 3.4 Mbps, bus loading = 100pF

160 ns for 3.4 Mbps, bus loading = 400pF

MIN_SCL_LOWtime = minimum low period

MIN_SCL_LOWtime =

4700 ns for 100 kbps

1300 ns for 400 kbps

120 ns for 3.4Mbps, bus loading = 100pF

320 ns for 3.4Mbps, bus loading = 400pF

$OSCFREQ$ = $\text{t}_{4\_sp\_clk}$ clock frequency (Hz)

Calculating High and Low Counts

$OSCFREQ = 100 \text{ MHz}$

$I2Cmode = \text{fast}, 400 \text{ kbps}$

$\text{MIN\_SCL\_HIGHTime} = 600 \text{ ns}$

$\text{MIN\_SCL\_LOWtime} = 1300 \text{ ns}$

```
IC_HCNT = ceil(600 ns * 100 MHz) IC_HCNTSCL_PERIOD = 60
IC_LCNT = ceil(1300 ns * 100 MHz) IC_LCNTSCL_PERIOD = 130
Actual MIN_SCL_HIGHTime = 60*(1/100 MHz) = 600 ns
Actual MIN_SCL_LOWtime = 130*(1/100 MHz) = 1300 ns †
```

You may configure the SCL and SDA falling time using the parameters `i2c-scl-falling-time-ns` and `i2c-sda-falling-time-ns` respectively.

SDA Hold Time

The I²C protocol specification requires 300 ns of hold time on the SDA signal in standard and fast speed modes. Board delays on the SCL and SDA signals can mean that the hold time requirement is met at the I²C master, but not at the I²C slave (or vice-versa). As each application encounters differing board delays, the I²C controller contains a software programmable register, `IC_SDA_HOLD`, to enable dynamic adjustment of the SDA hold time. `IC_SDA_HOLD` effects both slave-transmitter and master mode.

DMA Controller Interface

The I²C controller supports DMA signaling to indicate when data is ready to be read or when the transmit FIFO needs data. This support requires 2 DMA channels, one for transmit data and one for receive data. The I²C controller supports both single and burst DMA transfers. System software can choose the DMA burst mode by programming an appropriate value into the threshold registers. The recommended setting of the FIFO threshold register value is half full.

To enable the DMA controller interface on the I²C controller, you must write to the DMA control register (DMACR) bits. Writing a 1 into the `TDMAE` bit field of DMACR register enables the I²C controller transmit handshaking interface. Writing a 1 into the `RDMAE` bit field of the DMACR register enables the I²C controller receive handshaking interface. †

Related Information

[DMA Controller](#) on page 16-1

For details about the DMA burst length microcode setup, refer to the *DMA controller* chapter.

Clocks

Each I²C controller is connected to the `l4_sp_clk` clock, which clocks transfers in standard and fast mode. The clock input is driven by the clock manager.

Related Information

[Clock Manager](#) on page 3-1

For more information, refer to *Clock Manager* chapter.

Resets

Each I²C controller has a separate reset signal. The reset manager drives the signals on a cold or warm reset.

Related Information

[Reset Manager](#) on page 4-1

For more information, refer to *Reset Manager* chapter.

Taking the I²C Controller Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

I²C Controller Programming Model

This section describes the programming model for the I²C controllers based on the two master and slave operation modes. †

Note: Each I²C controller should be set to operate only as an I²C master or as an I²C slave, never set both simultaneously. Ensure that bit 6 (IC_SLAVE_DISABLE) and 0 (IC_MASTER_MODE) of the IC_CON register are never set to 0 and 1, respectively. †

Slave Mode Operation

Initial Configuration

To use the I²C controller as a slave, perform the following steps: †

1. Disable the I²C controller by writing a 0 to bit 0 of the IC_ENABLE register. †
2. Write to the IC_SAR register (bits 9:0) to set the slave address. This is the address to which the I²C controller responds. †

Note: The reset value for the I²C controller slave address is 0x55. If you are using 0x55 as the slave address, you can safely skip this step.

3. Write to the IC_CON register to specify which type of addressing is supported (7- or 10-bit by setting bit 3). Enable the I²C controller in slave-only mode by writing a 0 into bit 6 (IC_SLAVE_DISABLE) and a 0 to bit 0 (MASTER_MODE). †

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

4. Enable the I²C controller by writing a 1 in bit 0 of the IC_ENABLE register. †

Slave-Transmitter Operation for a Single Byte

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

1. The other I²C master device initiates an I²C transfer with an address that matches the slave address in the IC_SAR register of the I²C controller. †
2. The I²C controller acknowledges the sent address and recognizes the direction of the transfer to indicate that it is acting as a slave-transmitter. †
3. The I²C controller asserts the RD_REQ interrupt (bit 5 of the IC_RAW_INTR_STAT register) and waits for software to respond. †

If the RD_REQ interrupt has been masked, due to bit 5 of the IC_INTR_MASK register (M_RD_REQ bit field) being set to 0, then it is recommended that you instruct the CPU to perform periodic reads of the IC_RAW_INTR_STAT register. †

- Reads that indicate bit 5 of the IC_RAW_INTR_STAT register (R_RD_REQ bit field) being set to 1 must be treated as the equivalent of the RD_REQ interrupt being asserted. †
- Software must then act to satisfy the I²C transfer. †
- The timing interval used should be in the order of 10 times the fastest SCL clock period the I²C controller can handle. For example, for 400 Kbps, the timing interval is 25 us. †

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

4. If there is any data remaining in the TX FIFO before receiving the read request, the I²C controller asserts a TX_ABRT interrupt (bit 6 of the IC_RAW_INTR_STAT register) to flush the old data from the TX FIFO. †

Note: Because the I²C controller's TX FIFO is forced into a flushed/reset state whenever a TX_ABRT event occurs, it is necessary for software to release the I²C controller from this state by reading the IC_CLR_TX_ABRT register before attempting to write into the TX FIFO. For more information, refer to the IC_RAW_INTR_STAT register description in the register map. †

If the TX_ABRT interrupt has been masked, due to of IC_INTR_MASK[6] register (M_TX_ABRT bit field) being set to 0, then it is recommended that the CPU performs periodic reads of the IC_RAW_INTR_STAT register. †

- Reads that indicate bit 6 (R_TX_ABRT) being set to 1 must be treated as the equivalent of the TX_ABRT interrupt being asserted. †
 - There is no further action required from software. †
 - The timing interval used should be similar to that described in the previous step for the IC_RAW_INTR_STAT[5] register. †
5. Software writes to the DAT bits of the IC_DATA_CMD register with the data to be written and writes a 0 in bit 8. †
 6. Software must clear the RD_REQ and TX_ABRT interrupts (bits 5 and 6, respectively) of the IC_RAW_INTR_STAT register before proceeding. †

If the RD_REQ or TX_ABRT interrupt is masked, then clearing of the IC_RAW_INTR_STAT register has already been performed when either the R_RD_REQ or R_TX_ABRT bit has been read as 1.

7. The I²C controller transmits the byte. †
8. The master may hold the I²C bus by issuing a RESTART condition or release the bus by issuing a STOP condition. †

Slave-Receiver Operation for a Single Byte

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



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

Note: If the RX FIFO is completely filled with data when a byte is pushed, then an overflow occurs and the I²C controller continues with subsequent I²C transfers. Because a NACK is not generated, software must recognize the overflow when indicated by the I²C controller (by the R_RX_OVER bit in the IC_INTR_STAT register) and take appropriate actions to recover from lost data. Hence, there is a real time constraint on software to service the RX FIFO before the latter overflow as there is no way to reapply pressure to the remote transmitting master. †

4. I²C controller asserts the RX_FULL interrupt (IC_RAW_INTR_STAT[2] register). †

If the RX_FULL interrupt has been masked, due to setting IC_INTR_MASK[2] register to 0 or setting IC_TX_TL to a value larger than 0, then it is recommended that the CPU does periodic reads of the IC_STATUS register. Reads of the IC_STATUS register, with bit 3 (RFNE) set at 1, must then be treated by software as the equivalent of the RX_FULL interrupt being asserted. †

5. Software may read the byte from the IC_DATA_CMD register (bits 7:0). †
6. The other master device may hold the I²C bus by issuing a RESTART condition or release the bus by issuing a STOP condition. †

Slave-Transfer Operation for Bulk Transfers

In the standard I²C protocol, all transactions are single byte transactions and the programmer responds to a remote master read request by writing one byte into the slave's TX FIFO. When a slave (slave-transmitter) is issued with a read request (RD_REQ) from the remote master (master-receiver), at a minimum there should be at least one entry placed into the slave-transmitter's TX FIFO. The I²C controller is designed to handle more data in the TX FIFO so that subsequent read requests can receive that data without raising an interrupt to request more data. Ultimately, this eliminates the possibility of significant latencies being incurred between raising the interrupt for data each time had there been a restriction of having only one entry placed in the TX FIFO. †

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

If the RD_REQ interrupt is masked, due to bit 5 (M_RD_REQ) of the IC_INTR_STAT register being set to 0, then it is recommended that the CPU does periodic reads of the IC_RAW_INTR_STAT register. Reads of IC_RAW_INTR_STAT that return bit 5 (R_RD_REQ) set to 1 must be treated as the equivalent of the RD_REQ interrupt referred to in this section. †

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

If the programmer knows in advance that the remote master is requesting a packet of n bytes, then when another master addresses the I²C controller and requests data, the TX FIFO could be written with n number bytes and the remote master receives it as a continuous stream of data. For example, the I²C controller slave continues to send data to the remote master as long as the remote master is acknowledging the data sent and there is data available in the TX FIFO. There is no need to issue RD_REQ again. †

If the remote master is to receive n bytes from the I²C controller but the programmer wrote a number of bytes larger than n to the TX FIFO, then when the slave finishes sending the requested n bytes, it clears the TX FIFO and ignores any excess bytes. †

The I²C controller generates a transmit abort (TX_ABORT) event to indicate the clearing of the TX FIFO in this example. At the time an ACK/NACK is expected, if a NACK is received, then the remote master has all the data it wants. At this time, a flag is raised within the slave's state machine to clear the leftover data in the TX FIFO. This flag is transferred to the processor bus clock domain where the FIFO exists and the contents of the TX FIFO are cleared at that time. †

Master Mode Operation

Initial Configuration

For master mode operation, the target address and address format can be changed dynamically without having to disable the I²C controller. This feature is only applicable when the I²C controller is acting as a master because the slave requires the component to be disabled before any changes can be made to the address. To use the I²C controller as a master, perform the following steps: †

For multiple I²C transfers, perform additional writes to the Tx FIFO such that the Tx FIFO does not become empty during the I²C transaction. If the Tx FIFO is completely emptied at any stage, then the master stalls the transfer by holding the SCL line low because there was no stop bit indicating the master to issue a STOP. The master completes the transfer when it finds a Tx FIFO entry tagged with a Stop bit.

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

Dynamic IC_TAR or IC_10BITADDR_MASTER Update

The I²C controller supports dynamic updating of the IC_TAR (bits 9:0) and IC_10BITADDR_MASTER (bit 12) bit fields of the IC_TAR register. You can dynamically write to the IC_TAR register provided the following conditions are met: †

- The I²C controller is not enabled (IC_ENABLE=0); †
- The I²C controller is enabled (IC_ENABLE=1); AND I²C controller is NOT engaged in any Master (TX, RX) operation (IC_STATUS[5]=0); AND I²C controller is enabled to operate in Master mode (IC_CON[0]=1); AND there are no entries in the TX FIFO (IC_STATUS[2]=1) †

Master Transmit and Master Receive

The I²C controller supports switching back and forth between reading and writing dynamically. To transmit data, write the data to be written to the lower byte of the I²C Rx/Tx Data Buffer and Command Register (IC_DATA_CMD). The CMD bit [8] should be written to 0 for I²C write operations. Subsequently, a

read command may be issued by writing "don't cares" to the lower byte of the IC_DATA_CMD register, and a 1 should be written to the CMD bit.†

Disabling the I²C Controller

The register IC_ENABLE_STATUS is added to allow software to unambiguously determine when the hardware has completely shutdown in response to the IC_ENABLE register being set from 1 to 0. †

1. Define a timer interval (`ti2c_poll`) equal to the 10 times the signaling period for the highest I²C transfer speed used in the system and supported by the I²C controller. For example, if the highest I²C transfer mode is 400 Kbps, then `ti2c_poll` is 25 us. †
2. Define a maximum time-out parameter, `MAX_T_POLL_COUNT`, such that if any repeated polling operation exceeds this maximum value, an error is reported. †
3. Execute a blocking thread/process/function that prevents any further I²C master transactions to be started by software, but allows any pending transfers to be completed.
 - This step can be ignored if the I²C controller is programmed to operate as an I²C slave only. †
4. The variable `POLL_COUNT` is initialized to zero. †
5. Set `IC_ENABLE` to 0. †
6. Read the `IC_ENABLE_STATUS` register and test the `IC_EN` bit (bit 0). Increment `POLL_COUNT` by one. If `POLL_COUNT >= MAX_T_POLL_COUNT`, exit with the relevant error code. †
7. If `IC_ENABLE_STATUS[0]` is 1, then sleep for `ti2c_poll` and proceed to the previous step. Otherwise, exit with a relevant success code. †

DMA Controller Operation

To enable the DMA controller interface on the I²C controller, you must write the DMA Control Register (`IC_DMA_CR`). Writing a 1 to the `TDMAE` bit field of `IC_DMA_CR` register enables the I²C controller transmit handshaking interface. Writing a 1 to the `RDMAE` bit field of the `IC_DMA_CR` register enables the I²C controller receive handshaking interface.†

The FIFO buffer depth (`FIFO_DEPTH`) for both the RX and TX buffers in the I²C controller is 64 entries.

Related Information

[DMA Controller](#) on page 16-1

For details about the DMA burst length microcode setup, refer to the *DMA controller* chapter.

Transmit FIFO Underflow

During I²C serial transfers, transmit FIFO requests are made to the DMA controller whenever the number of entries in the transmit FIFO is less than or equal to the value in DMA Transmit Data Level Register (`IC_DMA_TDLR`), also known as the watermark level. The DMA controller responds by writing a burst of data to the transmit FIFO buffer, of length specified as DMA burst length. †

Note: Data should be fetched from the DMA often enough for the transmit FIFO to perform serial transfers continuously, that is, when the FIFO begins to empty, another DMA request should be triggered. Otherwise, the FIFO runs out of the data (underflow) causing the master to stall the transfer by holding the SCL line low. To prevent this condition, you must set the watermark level correctly.†

Related Information

[DMA Controller](#) on page 16-1

For details about the DMA burst length microcode setup, refer to the *DMA controller* chapter.

Transmit Watermark Level

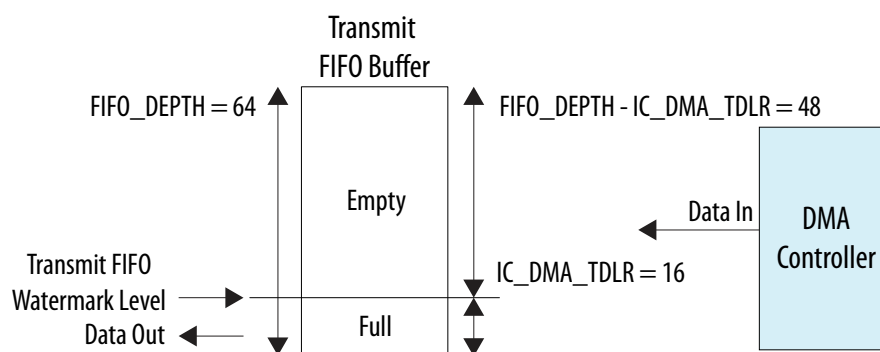
Consider the example where the assumption is made: †

$$\text{DMA burst length} = \text{FIFO\_DEPTH} - \text{IC\_DMA\_TDLR} \dagger$$

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

- Case 1: IC_DMA_TDLR = 16: †
 - Transmit FIFO watermark level = IC_DMA_TDLR = 16: †
 - DMA burst length = FIFO_DEPTH - IC_DMA_TDLR = 48: †
 - I²C transmit FIFO_DEPTH = 64: †
 - Block transaction size = 240: †

Figure 20-13: Transmit FIFO Watermark Level = 16



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

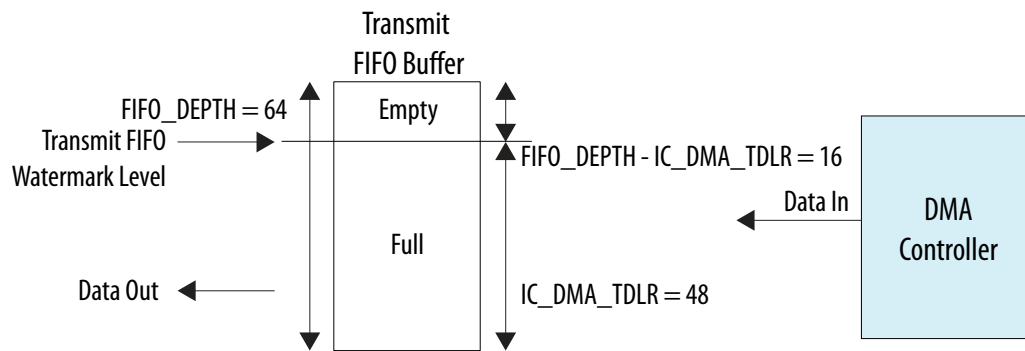
$$\text{Block transaction size/DMA burst length} = 240/48 = 5$$

The number of burst transactions in the DMA block transfer is 5. But the watermark level, IC_DMA_TDLR, is quite low. Therefore, the probability of transmit underflow is high where the I²C serial transmit line needs to transmit data, but 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 FIFO becomes empty.

- Case 2: IC_DMA_TDLR = 48 †
 - Transmit FIFO watermark level = IC_DMA_TDLR = 48 †
 - DMA burst length = FIFO_DEPTH - IC_DMA_TDLR = 16 †
 - I²C transmit FIFO_DEPTH = 64 †
 - Block transaction size = 240 †



Figure 20-14: Transmit FIFO Watermark Level = 48



Number of burst transactions in block: †

Block transaction size/DMA burst length = $240/16 = 15$ †

In this block transfer, there are 15 destination burst transactions in a DMA block transfer. But the watermark level, IC_DMA_TDLR, is high. Therefore, the probability of I²C transmit underflow is low because the DMA controller has plenty of time to service the destination burst transaction request before the I²C transmit FIFO becomes empty. †

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

Therefore, the goal in choosing a watermark level is to minimize the number of transactions per block, while at the same time keeping the probability of an underflow condition to an acceptable level. In practice, this is a function of the ratio of the rate at which the I²C transmits data to the rate at which the DMA can respond to destination burst requests. †

Transmit FIFO Overflow

Setting the DMA burst length to a value greater than the watermark level that triggers the DMA request might cause overflow when there is not enough space in the transmit FIFO to service the destination burst request. Therefore, the following equation must be adhered to in order to avoid overflow: †

$$\text{DMA burst length} \leq \text{FIFO\_DEPTH} - \text{IC\_DMA\_TDLR}$$

In case 2: IC_DMA_TDLR = 48, the amount of space in the transmit FIFO at the time of the burst request is made is equal to the DMA burst length. Thus, the transmit FIFO may be full, but not overflowed, at the completion of the burst transaction. †

Therefore, for optimal operation, DMA burst length should be set at the FIFO level that triggers a transmit DMA request; that is: †

$$\text{DMA burst length} = \text{FIFO\_DEPTH} - \text{IC\_DMA\_TDLR}$$

Adhering to this equation reduces the number of DMA bursts needed for block transfer, and this in turn improves bus utilization. †

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

Receive FIFO Overflow

During I²C serial transfers, receive FIFO requests are made to the DMA whenever the number of entries in the receive FIFO is at or above the DMA Receive Data Level Register, that is $IC\_DMA\_RDLR + 1$. This is known as the watermark level. The DMA responds by fetching a burst of data from the receive FIFO. †

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

Receive Watermark Level

Similar to choosing the transmit watermark level described earlier, the receive watermark level, $IC\_DMA\_RDLR + 1$, should be set to minimize the probability of overflow, as shown in the Receive FIFO Buffer diagram. It is a trade off between the number of DMA burst transactions required per block versus the probability of an overflow occurring. †

Receive FIFO Underflow

Setting the source transaction burst length greater than the watermark level can cause underflow where there is not enough data to service the source burst request. Therefore, the following equation must be adhered to avoid underflow: †

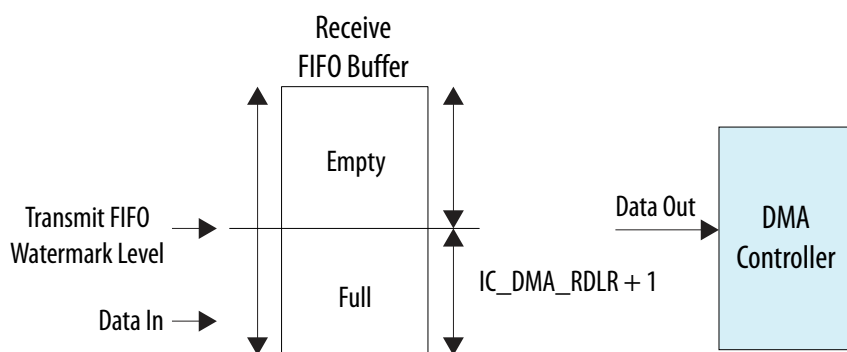
$$\text{DMA burst length} = IC\_DMA\_RDLR + 1$$

If the number of data items in the receive FIFO is equal to the source burst length at the time of the burst request is made, the receive FIFO may be emptied, but not underflowed, at the completion of the burst transaction. For optimal operation, DMA burst length should be set at the watermark level, $IC\_DMA\_RDLR + 1$. †

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

Note: The receive FIFO is not be empty at the end of the source burst transaction if the I²C controller has successfully received one data item or more on the I²C serial receive line during the burst. †

Figure 20-15: Receive FIFO Buffer



I²C Controller Address Map and Register Definitions

The address map and register definitions for the HPS-FPGA bridges consist of the following regions:

- I²C Module 0
- I²C Module 1
- I²C Module 2
- I²C Module 3

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
For more information, refer to *Introduction to the Hard Processor System* chapter.
- <https://www.intel.com/content/www/us/en/programmable/hps/arrria-v/hps.html>

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) provides two UART controllers for asynchronous serial communication. The UART controllers are based on an industry standard 16550 UART controller. The UART controllers are instances of the Synopsys DesignWare APB Universal Asynchronous Receiver/Transmitter (DW_apb_uart) peripheral.⁽⁵⁸⁾

UART Controller Features

The UART controller provides the following functionality and features:

- Programmable character properties, such as number of data bits per character, optional parity bits, and number of stop bits †
- Line break generation and detection †
- Direct memory access (DMA) controller interface
- Prioritized interrupt identification †
- Programmable baud rate
- False start bit detection †
- Automatic flow control mode per 16750 standard †
- Internal loopback mode support
- 128-byte transmit and receive FIFO buffers
 - FIFO buffer status registers †
 - FIFO buffer access mode (for FIFO buffer testing) enables write of receive FIFO buffer by master and read of transmit FIFO buffer by master †
- Shadow registers reduce software overhead and provide programmable reset †
- Transmitter holding register empty (THRE) interrupt mode †

⁽⁵⁸⁾ Portions © 2017 Synopsys, Inc. Used with permission. All rights reserved. Synopsys & DesignWare are registered trademarks of Synopsys, Inc. All documentation is provided "as is" and without any warranty. Synopsys expressly disclaims any and all warranties, express, implied, or otherwise, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, and any warranties arising out of a course of dealing or usage of trade.

†Paragraphs marked with the dagger (†) symbol are Synopsys Proprietary. Used with permission.

UART Controller Block Diagram and System Integration

Figure 21-1: UART Block Diagram

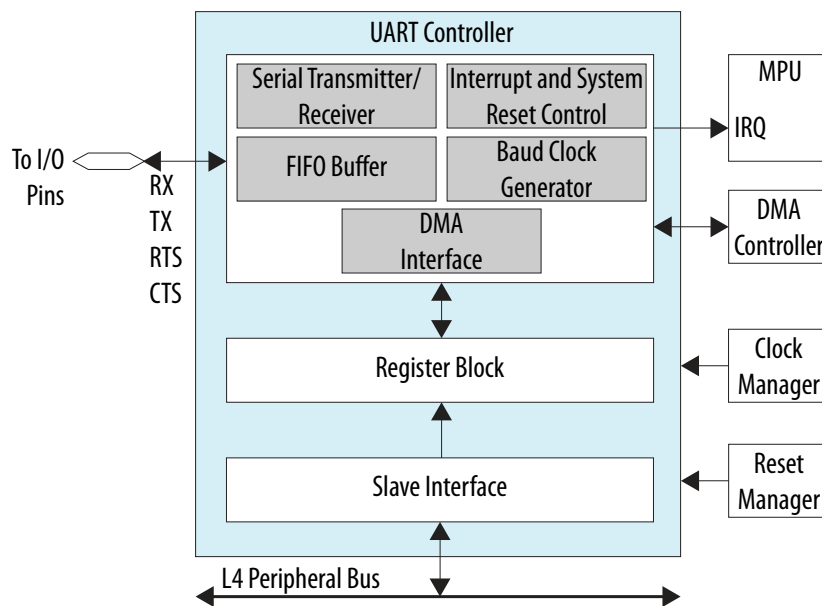


Table 21-1: UART Controller Block Descriptions

Block	Description
Slave interface	Slave interface between the component and L4 peripheral bus.
Register block	Provides main UART control, status, and interrupt generation functions.†
FIFO buffer	Provides FIFO buffer control and storage. †
Baud clock generator	Generates the transmitter and receiver baud clock. With a reference clock of 100 MHz, the UART controller supports transfer rates of 95 baud to 6.25 Mbaud. This supports communication with all known 16550 devices. The baud rate is controlled by programming the interrupt enable or divisor latch high (IER_DLH) and receive buffer, transmit holding, or divisor latch low (RBR_THR_DLL) registers.
Serial transmitter	Converts parallel data written to the UART into serial data and adds all additional bits, as specified by the control register, for transmission. This makeup of serial data, referred to as a character, exits the block in serial UART. †

Block	Description
Serial receiver	Converts the serial data character (as specified by the control register) received in the UART format to parallel form. Parity error detection, framing error detection and line break detection is carried out in this block. †
DMA interface	The UART controller includes a DMA controller interface to indicate when received data is available or when the transmit FIFO buffer requires data. The DMA requires two channels, one for transmit and one for receive. The UART controller supports single and burst transfers. You can use DMA in FIFO buffer and non-FIFO buffer mode.

Related Information[DMA Controller](#) on page 16-1

For more information, refer to the DMA Controller chapter.

UART Controller Signal Description

HPS I/O Pins

Table 21-2: HPS I/O UART Pin Descriptions

Pin	Width	Direction	Description
RX	1 bit	Input	Serial Input
TX	1 bit	Output	Serial Output
CTS	1 bit	Input	Clear to send
RTS	1 bit	Output	Request to send

FPGA Routing

Table 21-3: Signals for FPGA Routing

Signal	Width	Direction	Description
uart_rxd	1 bit	Input	Serial input
uart_txd	1 bit	Output	Serial output



Signal	Width	Direction	Description
uart_cts	1 bit	Input	Clear to send
uart_rts	1 bit	Output	Request to send
uart_dsr	1 bit	Input	Data set ready
uart_dcd	1 bit	Input	Data carrier detect
uart_ri	1 bit	Input	Ring indicator
uart_dtr	1 bit	Output	Data terminal ready
uart_out1_n	1 bit	Output	User defined output 1
uart_out2_n	1 bit	Output	User defined output 2

Functional Description of the UART Controller

The HPS UART is based on an industry-standard 16550 UART. The UART supports serial communication with a peripheral, modem (data carrier equipment), or data set. The master (CPU) writes data over the slave bus to the UART. The UART converts the data to serial format and transmits to the destination device. The UART also receives serial data and stores it for the master (CPU). †

The UART's registers control the character length, baud rate, parity generation and checking, and interrupt generation. The UART's single interrupt output signal is supported by several prioritized interrupt types that trigger assertion. You can separately enable or disable each of the interrupt types with the control registers. †

FIFO Buffer Support

The UART controller includes 128-byte FIFO buffers to buffer transmit and receive data. FIFO buffer access mode allows the master to write the receive FIFO buffer and to read the transmit FIFO buffer for test purposes. FIFO buffer access mode is enabled with the FIFO access register (FAR). Once enabled, the control portions of the transmit and receive FIFO buffers are reset and the FIFO buffers are treated as empty. †

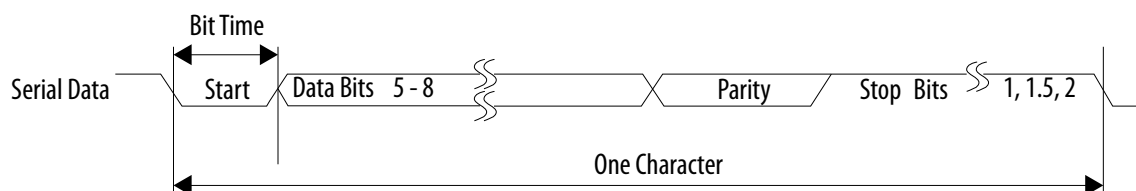
When FIFO buffer access mode is enabled, you can write data to the transmit FIFO buffer as normal; however, no serial transmission occurs in this mode and no data leaves the FIFO buffer. You can read back the data that is written to the transmit FIFO buffer with the transmit FIFO read (TFR) register. The TFR register provides the current data at the top of the transmit FIFO buffer. †

Similarly, you can also read data from the receive FIFO buffer in FIFO buffer access mode. Since the normal operation of the UART is halted in this mode, you must write data to the receive FIFO buffer to read it back. The receive FIFO write (RFW) register writes data to the receive FIFO buffer. The upper two bits of the 10-bit register write framing errors and parity error detection information to the receive FIFO buffer. Bit 9 of RFW indicates a framing error and bit 8 of RFW indicates a parity error. Although you cannot read these bits back from the receive buffer register, you can check the bits by reading the line status register (LSR), and by checking the corresponding bits when the data in question is at the top of the receive FIFO buffer. †

UART(RS232) Serial Protocol

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

Figure 21-2: Serial Data Format



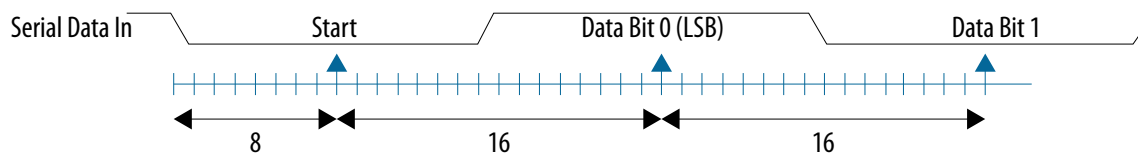
An additional parity bit may be added to the serial character. This bit appears after the last data bit and before the stop bit(s) in the character structure to provide the UART controller with the ability to perform simple error checking on the received data.†

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

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

Together with serial input debouncing, this feature also contributes to avoid the detection of false start bits. Short glitches are filtered out by debouncing, and no transition is detected on the line. If a glitch is wide enough to avoid filtering by debouncing, a falling edge is detected. However, a start bit is detected only if the line is sampled low again after half a bit time has elapsed. †

Figure 21-3: Receiver Serial Data Sample Points



The baud rate of the UART controller is controlled by the serial clock and the Divisor Latch Register (DLH and DLL).†

Automatic Flow Control

The UART includes 16750-compatible request-to-send (RTS) and clear-to-send (CTS) serial data automatic flow control mode. You enable automatic flow control with the modem control register (MCR.AFCE). †

Automatic RTS mode

Automatic RTS mode becomes active when the following conditions occur: †

- RTS (MCR.RTS bit and MCR.AFCE bit are both set)
- FIFO buffers are enabled (FCR.FIFOE bit is set)

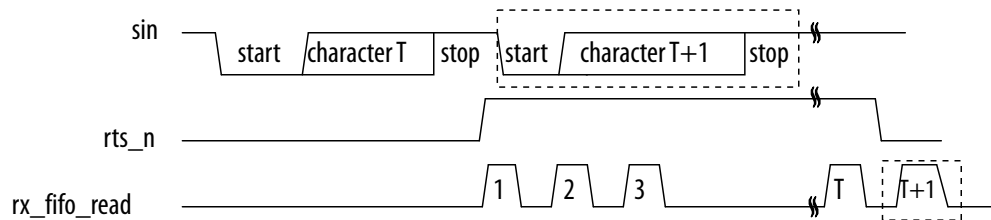
When `rts_n` is connected to the `cts_n` input pin of another UART device, the other UART stops sending serial data until the receive FIFO buffer has available space (until it is completely empty). †

The selectable receive FIFO buffer threshold values are 1, $\frac{1}{4}$, $\frac{1}{2}$, and 2 less than full. Because one additional character may be transmitted to the UART after `rts_n` is inactive (due to data already having entered the transmitter block in the other UART), setting the threshold to 2 less than full allows maximum use of the FIFO buffer with a margin of one character. †

Once the receive FIFO buffer is completely emptied by reading the receiver buffer register (RBR_THR_DLL), `rts_n` again becomes active (low), signaling the other UART to continue sending data. †

Even when you set the correct MCR bits, if the FIFO buffers are disabled through FCR.FIFOE, automatic flow control is also disabled. When auto RTS is not implemented or disabled, `rts_n` is controlled solely by MCR.RTS. In the Automatic RTS Timing diagram, the character T is received because `rts_n` is not detected prior to the next character entering the sending UART transmitter. †

Figure 21-4: Automatic RTS Timing



Automatic CTS mode

Automatic CTS mode becomes active when the following conditions occur: †

- AFCE (MCR.AFCE bit is set)
- FIFO buffers are enabled (through FIFO buffer control register IIR_FCR.FIFOE) bit

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

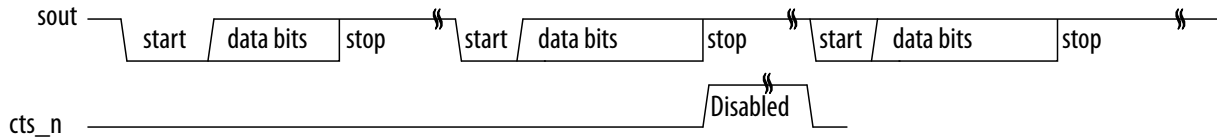
If the `cts_n` input is not deactivated before the middle of the last stop bit, another character is transmitted before the transmitter is disabled. While the transmitter is disabled, you can continue to write and even overflow to the transmit FIFO buffer. †

Automatic CTS mode requires the following sequence:

1. The UART status register are read to verify that the transmit FIFO buffer is full (UART status register USR.TFNF set to zero). †
2. The current FIFO buffer level is read via the transmit FIFO level (TFL) register. †
3. Programmable THRE interrupt mode must be enabled to access the FIFO buffer full status from the LSR. †

When using the FIFO buffer full status, software can poll this before each write to the transmit FIFO buffer. When the `cts_n` input becomes active (low) again, transmission resumes. If the FIFO buffers are disabled with the `FCR.FIFOE` bit, automatic flow control is also disabled regardless of any other settings. When auto CTS is not implemented or disabled, the transmitter is unaffected by `cts_n`.†

Figure 21-5: Automatic CTS Timing



Clocks

The UART controller is connected to the `l4_sp_clk` clock. The clock input is driven by the clock manager.

Related Information

[Clock Manager](#) on page 3-1

For more information, refer to the *Clock Manager* chapter.

Resets

The UART controller is connected to the `uart_rst_n` reset signal. The reset manager drives the signal on a cold or warm reset.

Taking the UART Controller Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

Related Information

[Reset Manager](#) on page 4-1

For more information, refer to the *Reset Manager* chapter.

Interrupts

The assertion of the UART interrupt output signal occurs when one of the following interrupt types are enabled and active: †

Table 21-4: Interrupt Types and Priority †

Interrupt Type	Priority	Source	Interrupt Reset Control
Receiver line status	Highest	Overrun, parity and framing errors, break condition.	Reading the line status Register.
Received data available	Second	Receiver data available (FIFOs disabled) or RCVR FIFO trigger level reached (FIFOs enabled).	Reading the receiver buffer register (FIFOs disabled) or the FIFO drops below the trigger level (FIFOs enabled)
Character timeout indication	Second	No characters in or out of the Receive FIFO during the last 4 character times and there is at least 1 character in it during this Time.	Reading the receiver buffer Register.
Transmit holding register empty	Third	Transmitter holding register empty (Programmable THRE Mode disabled) or Transmit FIFO at or below threshold (Programmable THRE Mode enabled).	Reading the IIR register (if source of interrupt); or, writing into THR (FIFOs or Programmable THRE Mode not enabled) or Transmit FIFO above threshold (FIFOs and Programmable THRE Mode enabled).
Modem Status	Fourth	Clear to send or data set ready or ring indicator or data carrier detect. If auto flow control mode is enabled, a change in CTS (that is, DCTS set) does not cause an interrupt.	Reading the Modem status Register.

You can enable the interrupt types with the interrupt enable register (IER_DLH).

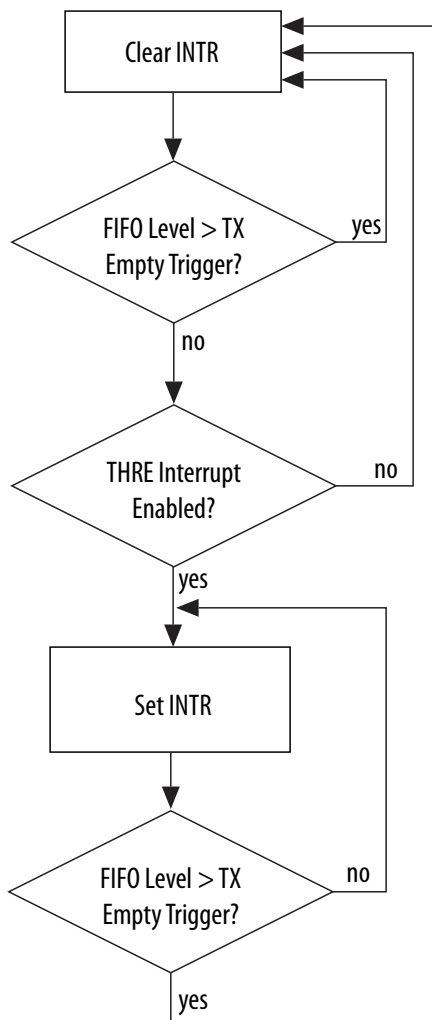
Note: "Received Data Available" and "Character Timeout Indication" are enabled by a single bit in the IER_DLH register, because they have the same priority.

Once an interrupt is signaled, you can determine the interrupt source by reading the Interrupt Identity Register (IIR).

Programmable THRE Interrupt

The UART has a programmable THRE interrupt mode to increase system performance. You enable the programmable THRE interrupt mode with the interrupt enable register (IER_DLH.PTIME). When the THRE mode is enabled, THRE interrupts and the dma_tx_req signal are active at and below a programmed transmit FIFO buffer empty threshold level, as shown in the flowchart. †

Figure 21-6: Programmable THRE Interrupt

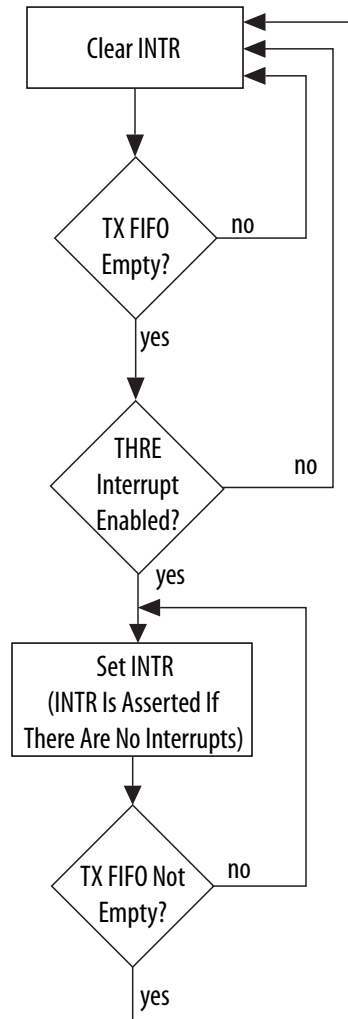


The threshold level is programmed into `FCR.TET`. The available empty thresholds are empty, 2, $\frac{1}{4}$, and $\frac{1}{2}$. The optimum threshold value depends on the system's ability to begin a new transmission sequence in a timely manner. However, one of these thresholds should prove optimum in increasing system performance by preventing the transmit FIFO buffer from running empty.

In addition to the interrupt change, line status register (`LSR.THRE`) also switches from indicating that the transmit FIFO buffer is empty, to indicating that the FIFO buffer is full. This change allows software to fill the FIFO buffer for each transmit sequence by polling `LSR.THRE` before writing another character. This directs the UART to fill the transmit FIFO buffer whenever an interrupt occurs and there is data to transmit, instead of waiting until the FIFO buffer is completely empty. Waiting until the FIFO buffer is empty reduces performance whenever the system is too busy to respond immediately. You can increase system efficiency when this mode is enabled in combination with automatic flow control.

When not selected or disabled, THRE interrupts and `LSR.THRE` function normally, reflecting an empty THRE or FIFO buffer.

Figure 21-7: Interrupt Generation without Programmable THRE Interrupt Mode



DMA Controller Operation

The UART controller includes a DMA controller interface to indicate when the receive FIFO buffer data is available or when the transmit FIFO buffer requires data. The DMA requires two channels, one for transmit and one for receive. The UART controller supports both single and burst transfers.

The FIFO buffer depth (FIFO_DEPTH) for both the RX and TX buffers in the UART controller is 128 entries.

Related Information

[DMA Controller](#) on page 16-1

For more information, refer to the DMA Controller chapter.

Transmit FIFO Underflow

During UART serial transfers, transmit FIFO requests are made to the DMA controller whenever the number of entries in the transmit FIFO is less than or equal to the decoded level of the Transmit Empty Trigger (TET) field in the FIFO Control Register (FCR), also known as the watermark level. The DMA controller responds by writing a burst of data to the transmit FIFO buffer, of length specified as DMA burst length. †

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

Related Information

[DMA Controller](#) on page 16-1

For more information, refer to the DMA Controller chapter.

Transmit Watermark Level

Consider the example where the following assumption is made: †

DMA burst length = FIFO_DEPTH - decoded watermark level of IIR_FCR.TET †

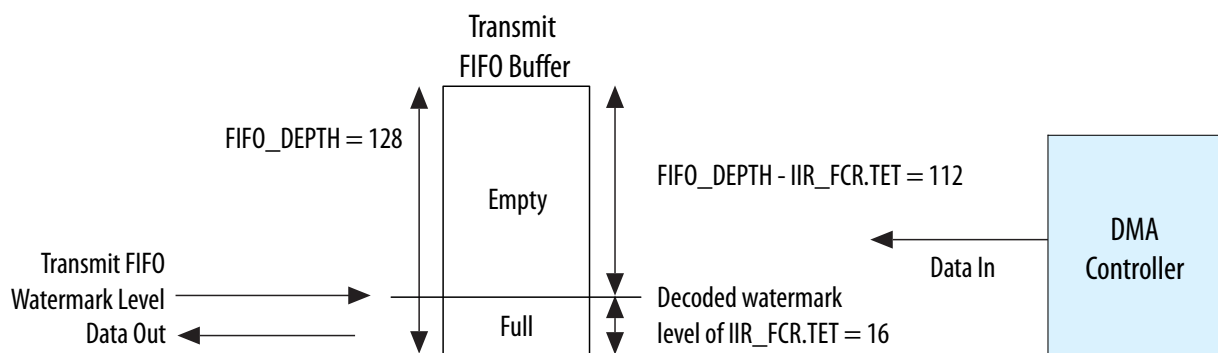
Here the number of data items to be transferred in a DMA burst is equal to the empty space in the transmit FIFO. Consider the following two different watermark level settings: †

IIR_FCR.TET = 1

IIR_FCR.TET = 1 decodes to a watermark level of 16.

- Transmit FIFO watermark level = decoded watermark level of IIR_FCR.TET = 16 †
- DMA burst length = FIFO_DEPTH - decoded watermark level of IIR_FCR.TET = 112 †
- UART transmit FIFO_DEPTH = 128 †
- Block transaction size = 448†

Figure 21-8: Transmit FIFO Watermark Level = 16



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

$$\text{Block transaction size/DMA burst length} = 448/112 = 4$$

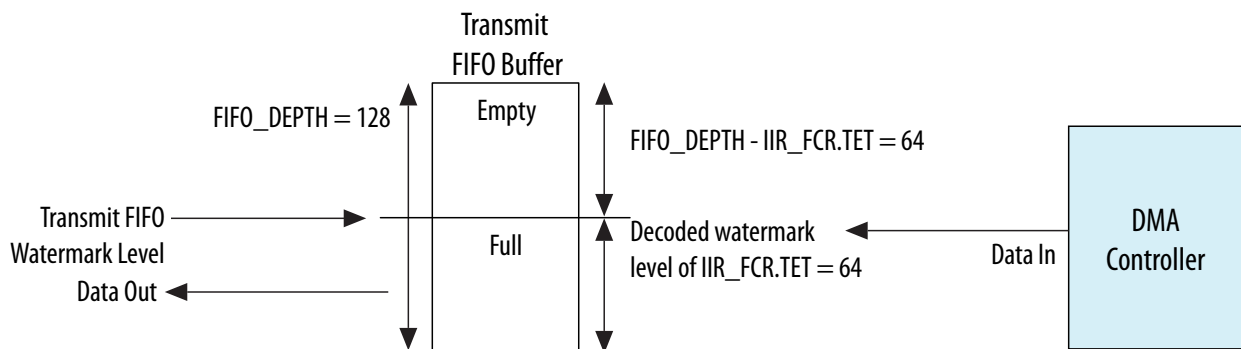
The number of burst transactions in the DMA block transfer is 4. But the watermark level, decoded level of IIR_FCR.TET, is quite low. Therefore, the probability of transmit underflow is high where the UART serial transmit line needs to transmit data, but there is no data left in the transmit FIFO. This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

IIR_FCR.TET = 3

IIR_FCR.TET = 3 decodes to a watermark level of 64.

- Transmit FIFO watermark level = decoded watermark level of IIR_FCR.TET = 64 †
- DMA burst length = FIFO_DEPTH - decoded watermark level of IIR_FCR.TET = 64 †
- UART transmit FIFO_DEPTH = 128 †
- Block transaction size = 448 †

Figure 21-9: Transmit FIFO Watermark Level = 64



Number of burst transactions in block: †

Block transaction size/DMA burst length = $448/64 = 7$ †

In this block transfer, there are 15 destination burst transactions in a DMA block transfer. But the watermark level, decoded level of IIR_FCR.TET, is high. Therefore, the probability of UART transmit underflow is low because the DMA controller has plenty of time to service the destination burst transaction request before the UART transmit FIFO becomes empty. †

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

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

Transmit FIFO Overflow

Setting the DMA burst length to a value greater than the watermark level that triggers the DMA request might cause overflow when there is not enough space in the transmit FIFO to service the destination burst request. Therefore, the following equation must be adhered to in order to avoid overflow: †

DMA burst length $\leq$ FIFO_DEPTH - decoded watermark level of IIR_FCR.TET

In case 2: decoded watermark level of `IIR_FCR.TET` = 64, the amount of space in the transmit FIFO at the time of the burst request is made is equal to the DMA burst length. Thus, the transmit FIFO may be full, but not overflowed, at the completion of the burst transaction. †

Therefore, for optimal operation, DMA burst length must be set at the FIFO level that triggers a transmit DMA request; that is: †

$$\text{DMA burst length} = \text{FIFO\_DEPTH} - \text{decoded watermark level of } \text{IIR\_FCR.TET}$$

Adhering to this equation reduces the number of DMA bursts needed for block transfer, and this in turn improves bus utilization. †

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

Receive FIFO Overflow

During UART serial transfers, receive FIFO requests are made to the DMA whenever the number of entries in the receive FIFO is at or above the decoded level of Receive Trigger (RT) field in the FIFO Control Register (`IIR_FCR`). This is known as the watermark level. The DMA responds by fetching a burst of data from the receive FIFO. †

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

Receive Watermark Level

Similar to choosing the transmit watermark level described earlier, the receive watermark level, decoded watermark level of `IIR_FCR.RT`, should be set to minimize the probability of overflow, as shown in the Receive FIFO Buffer diagram. It is a tradeoff between the number of DMA burst transactions required per block versus the probability of an overflow occurring. †

Receive FIFO Underflow

Setting the source transaction burst length greater than the watermark level can cause underflow where there is not enough data to service the source burst request. Therefore, the following equation must be adhered to avoid underflow: †

$$\text{DMA burst length} = \text{decoded watermark level of } \text{IIR\_FCR.RT} + 1$$

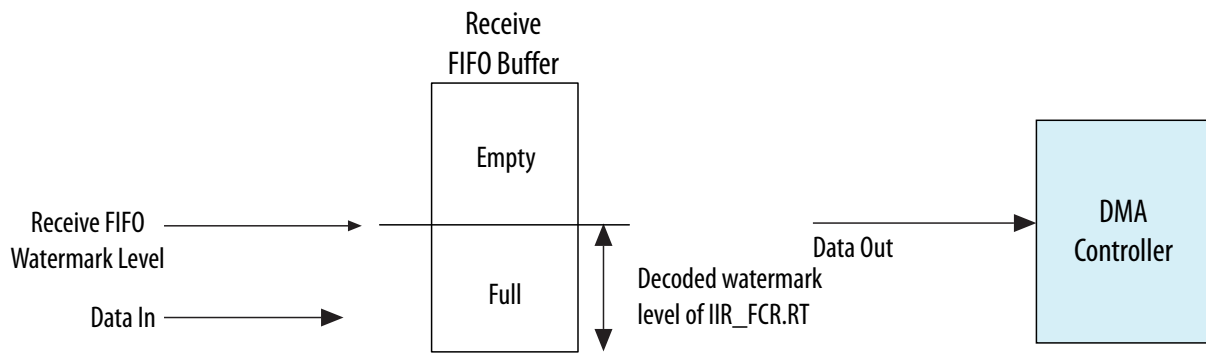
If the number of data items in the receive FIFO is equal to the source burst length at the time of the burst request is made, the receive FIFO may be emptied, but not underflowed, at the completion of the burst transaction. For optimal operation, DMA burst length should be set at the watermark level, decoded watermark level of `IIR_FCR.RT`. †

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

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



Figure 21-10: Receive FIFO Buffer



UART Controller Address Map and Register Definitions

The address map and register definitions for the HPS-FPGA bridges consist of the following regions:

- UART Module 0
- UART Module 1

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
For more information, refer to the *Introduction to the Hard Processor System* chapter.
- <https://www.intel.com/content/www/us/en/programmable/hps/arria-v/hps.html>

General-Purpose I/O Interface 22

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) provides three general-purpose I/O (GPIO) interface modules. The GPIO modules are instances of the Synopsys DesignWare APB General Purpose Programming I/O (DW_apb_gpio) peripheral.[†] ⁽⁵⁹⁾

Features of the GPIO Interface

The GPIO interface offers the following features:

- Supports digital debounce
- Configurable interrupt mode
- Supports up to 71 I/O pins and 14 input-only pins

⁽⁵⁹⁾ Portions © 2017 Synopsys, Inc. Used with permission. All rights reserved. Synopsys & DesignWare are registered trademarks of Synopsys, Inc. All documentation is provided "as is" and without any warranty. Synopsys expressly disclaims any and all warranties, express, implied, or otherwise, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, and any warranties arising out of a course of dealing or usage of trade.

[†]Paragraphs marked with the dagger (†) symbol are Synopsys Proprietary. Used with permission.

GPIO Interface Block Diagram and System Integration

The figure below shows a block diagram of the GPIO interface. The following table shows a pin table of the GPIO interface:

Figure 22-1: Arria V SoC GPIO

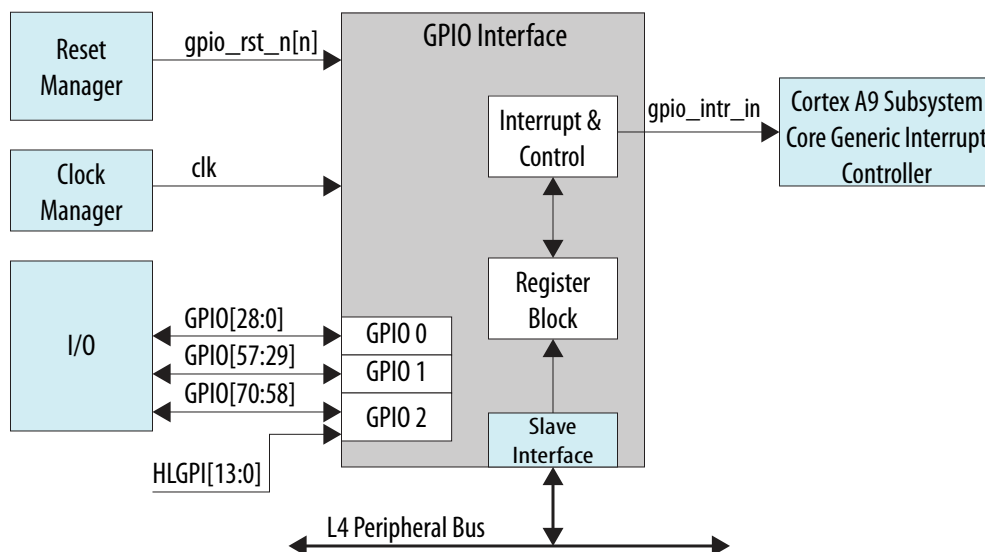


Table 22-1: GPIO Interface pin table

Pin Name	Mapped to GPIO Signal Name	Comments
GPIO [28:0]	GPIO 0 [28:0]	Input / Output
GPIO [57:29]	GPIO 1 [28:0]	Input / Output
GPIO [70:58]	GPIO 2 [12:0]	Input / Output
HLGPI [13:0]	GPIO 2 [26:13]	Input only

Related Information

[Arria V Device Handbook Volume 1: Device Interfaces and Integration](#)

For more information about I/O banks locations on the Arria V device, refer to the *I/O Banks Locations in Arria V Devices* section.

Functional Description of the GPIO Interface

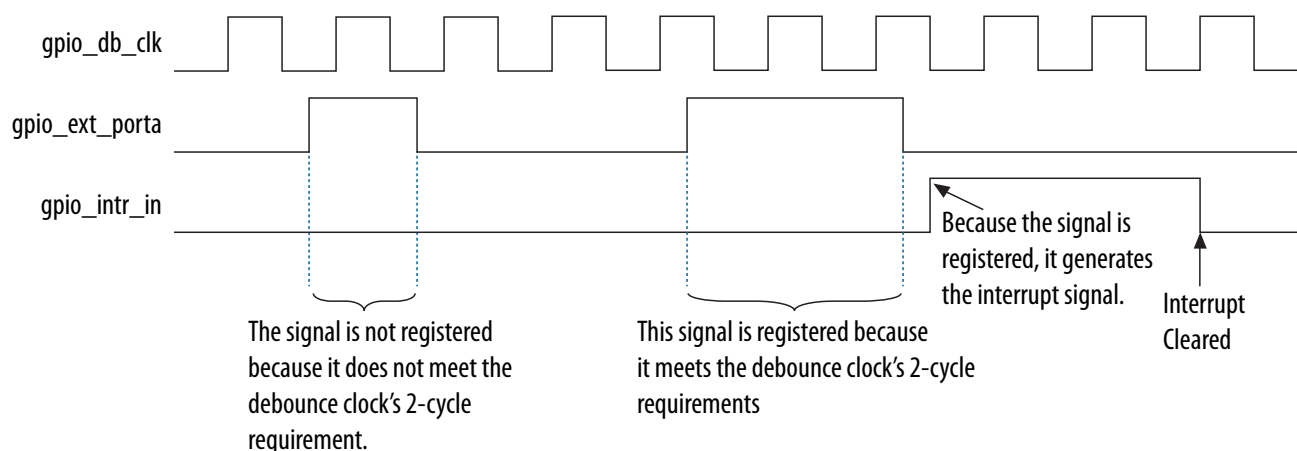
Debounce Operation

The GPIO modules provided in the HPS include optional debounce capabilities. The external signal can be debounced to remove any spurious glitches that are less than one period of the external debouncing clock, gpio_db_clk. †

When input signals are debounced using the `gpio_db_clk` debounce clock, the signals must be active for a minimum of two cycles of the debounce clock to guarantee that they are registered. Any input pulse widths less than a debounce clock period are filtered out. If the input signal pulse width is between one and two debounce clock widths, it may or may not be filtered out, depending on its phase relationship to the debounce clock. If the input pulse spans two rising edges of the debounce clock, it is registered. If it spans only one rising edge, it is not registered. †

The figure below shows a timing diagram of the debounce circuitry for both cases: a bounced input signal, and later, a propagated input signal.

Figure 22-2: Debounce Timing With Asynchronous Reset Flip-Flops



Note: Enabling the debounce circuitry increases interrupt latency by two clock cycles of the debounce clock.

Pin Directions

The pins `GPIO00` through `GPIO70` can be configured to be either input or output signals. The pins `HLGPIO0` through `HLGPIO13` share pins with the HPS DDR controller and are input-only signals.

Taking the GPIO Interface Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

Related Information

[Module Reset Signals](#) on page 4-5

GPIO Pin State During Reset

Ensure you understand the state of the GPIO during a warm or cold reset.

Cold reset: HPS I/O are in tri-state with a weak pull-up. The pin multiplexer and the GPIO/LoanIO multiplexer are in the default state meaning they are set to their reset value. You can find the default, reset value in the registers of the Pin Mux Control Group.

Warm reset: HPS I/O retain their configuration. The pin multiplexer and GPIO/LoanIO Multiplexer do not change configuration during warm reset meaning they are not reset to the default, reset value but maintain the current settings. For example, the pin multiplexer determines which signals go to the pins. Thus, if a pin multiplexer is set to be a GPIO it remains as a GPIO after warm reset. And when warm reset is triggered, the GPIO block is reset and the GPIO data direction register (`gpio_swporta_ddr`) is reset to 0x0 which is input, meaning the HPS I/O are inputs. In summary, a warm reset retains the pin multiplexer configuration but not the direction and output level which is controlled by the GPIO block. This block is reset under warm reset.

GPIO Interface Programming Model

Debounce capability for each of the input signals can be enabled or disabled under software control by setting the corresponding bits in the `gpio_debounce` register, accordingly. The debounce clock must be stable and operational before the debounce capability is enabled.

Under software control, the direction of the external I/O pad is controlled by a write to the `gpio_swportx_ddr` register. When configured as input mode, reading `gpio_ext_porta` would read the values on the signal of the external I/O pad. When configured as output mode, the data written to the `gpio_swporta_dr` register drives the output buffer of the I/O pad. The same pins are shared for both input and output modes, so they cannot be configured as input and output modes at the same time. †

General-Purpose I/O Interface Address Map and Register Definitions

The address map and register definitions for the GPIO consist of the following regions:

- GPIO Module 0
- GPIO Module 1
- GPIO Module 2

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
For more information, refer to *Introduction to the Hard Processor System* chapter.
- <https://www.intel.com/content/www/us/en/programmable/hps/arria-v/hps.html>



2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) provides four 32-bit general-purpose timers connected to the level 4 (L4) peripheral bus. The timers optionally generate an interrupt when the 32-bit binary count-down timer reaches zero. The timers are instances of the Synopsys DesignWare APB Timers (DW_apb_timers) peripheral.⁽⁶⁰⁾

Related Information

Cortex-A9 Microprocessor Unit Subsystem

The MPU subsystem provides additional timers. For more information about the timers in the MPU, refer to the *Cortex-A9 MPU* chapter.

Features of the Timer

- Supports interrupt generation
- Supports free-running mode
- Supports user-defined count mode

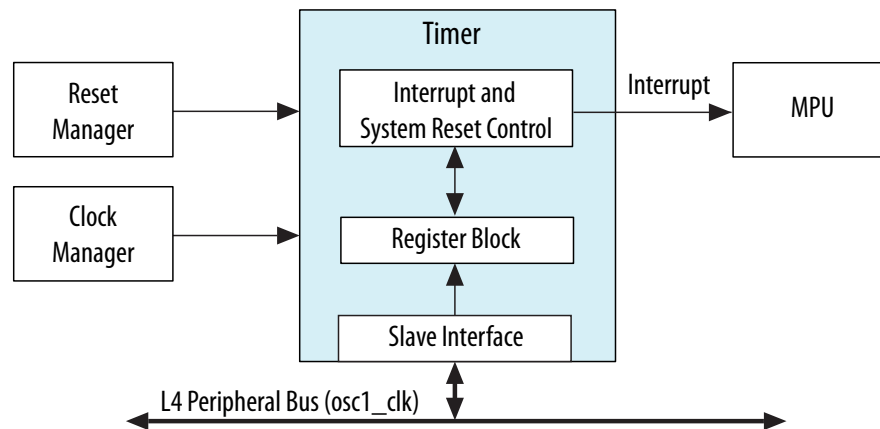
Timer Block Diagram and System Integration

Each timer includes a slave interface for control and status register (CSR) access, a register block, and a programmable 32-bit down counter that generates interrupts on reaching zero. The timer operates on a single clock domain driven by the clock manager.

⁽⁶⁰⁾ Portions © 2017 Synopsys, Inc. Used with permission. All rights reserved. Synopsys & DesignWare are registered trademarks of Synopsys, Inc. All documentation is provided "as is" and without any warranty. Synopsys expressly disclaims any and all warranties, express, implied, or otherwise, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, and any warranties arising out of a course of dealing or usage of trade.

Paragraphs marked with the dagger (†) symbol are Synopsys Proprietary. Used with permission.

Figure 23-1: Timer Block Diagram



Functional Description of the Timer

The 32-bit timer counts down from a programmed value and generates an interrupt when the count reaches zero. The timer has an independent clock input connected to the system clock signal or to an external clock source. †

The timer supports the following modes of operation:

- Free-running mode—decrementing from the maximum value (0xFFFFFFFF). Reloads maximum value upon reaching zero.
- User-defined count mode—generates a periodic interrupt. Decrements from the user-defined count value loaded from the timer1 load count register (timer1loadcount). Reloads the user-defined count upon reaching zero.

The initial value for the timer (that is, the value from which it counts down) is loaded into the timer by the timer1loadcount register. The following events can cause a timer to load the initial count from the timer1loadcount register: †

- Timer is enabled after being reset or disabled
- Timer counts down to 0

Clocks

Table 23-1: Timer Clock Characteristics

Timer	System Clock	Notes
OSC1 timer 0	osc1_clk	—
OSC1 timer 1		
SP timer 0	l4_sp_clk	Timer must be disabled if clock frequency changes
SP timer 1		

The timers above are labeled according to the clock it receives. OSC timers receive the oscillator clock `osc1_clk` and the SP timers receives the l4 slave peripheral clock `l4_sp_clk`.

SP timer 0 and SP timer 1 must be disabled before `l4_sp_clk` is changed to another frequency. You can then re-enable the timer once the clock frequency change takes effect. You cannot change the frequency of OSC1 timer 0 and OSC1 timer 1.

Related Information

[Clock Manager](#) on page 3-1

For more information about clock performance, refer to the *Clock Manager* chapter.

Resets

The timers are reset by a cold or warm reset. Resetting the timers produces the following results in the following order:

1. The timer is disabled.
2. The interrupt is enabled.
3. The timer enters free-running mode.
4. The timer count load register value is set to zero.

Interrupts

The timer1 interrupt status (`timer1intstat`) and timer1 end of interrupt (`timer1eoi`) registers handle the interrupts. The `timer1intstat` register allows you to read the status of the interrupt. Reading from the `timer1eoi` register clears the interrupt. †

The timer1 control register (`timer1controlreg`) contains the timer1 interrupt mask bit (`timer1_interrupt_mask`) to mask the interrupt. In both the free-running and user-defined count modes of operation, the timer generates an interrupt signal when the timer count reaches zero and the interrupt mask bit of the control register is high.

If the timer interrupt is set, then it is cleared when the timer is disabled.

FPGA Interface

The timer interrupts can be routed to the FPGA interface. You can configure and route the interrupts when you instantiate the HPS component in Platform Designer (Standard).

Related Information

[Interrupts](#) on page 26-5

For more information about configuring and routing timer interrupts, refer to the interrupt section of the *Instantiating the HPS Component* chapter in the *Hard Processor System Technical Reference Manual*.

Timer Programming Model

Initialization

To initialize the timer, perform the following steps: †

1. Initialize the timer through the `timer1controlreg` register: †

- Disable the timer by writing a 0 to the `timer1_enable` bit (`timer1_enable`) of the `timer1controlreg` register. †

Note: Before writing to a `timer1loadcount` register (`timer1loadcount`), you must disable the timer by writing a 0 to the `timer1_enable` bit of the `timer1controlreg` register to avoid potential synchronization problems. †

- Program the timer mode—user-defined count or free-running—by writing a 0 or 1, respectively, to the `timer1_mode` bit (`timer1_mode`) of the `timer1controlreg` register. †
 - Set the interrupt mask as either masked or not masked by writing a 1 or 0, respectively, to the `timer1_interrupt_mask` bit of the `timer1controlreg` register. †
2. Load the timer counter value into the `timer1loadcount` register. †
 3. Enable the timer by writing a 1 to the `timer1_enable` bit of the `timer1controlreg` register. †

Enabling the Timer

When a timer transitions to the enabled state, the current value of `timer1loadcount` register is loaded into the timer counter. †

1. To enable the timer, write a 1 to the `timer1_enable` bit of the `timer1controlreg` register.

Disabling the Timer

When the timer enable bit is cleared to 0, the timer counter and any associated registers in the timer clock domain, are asynchronously reset. †

1. To disable the timer, write a 0 to the `timer1_enable` bit. †

Loading the Timer Countdown Value

When a timer counter is enabled after being reset or disabled, the count value is loaded from the `timer1loadcount` register; this occurs in both free-running and user-defined count modes. †

When a timer counts down to 0, it loads one of two values, depending on the timer operating mode: †

- User-defined count mode—timer loads the current value of the `timer1loadcount` register. Use this mode if you want a fixed, timed interrupt. Designate this mode by writing a 1 to the `timer1_mode` bit of the `timer1controlreg` register. †
- Free-running mode—timer loads the maximum value (0xFFFFFFFF). The timer max count value allows for a maximum amount of time to reprogram or disable the timer before another interrupt occurs. Use this mode if you want a single timed interrupt. Enable this mode by writing a 0 to the `timer1_mode` bit of the `timer1controlreg` register. †

Servicing Interrupts

Clearing the Interrupt

An active timer interrupt can be cleared in two ways.

1. If you clear the interrupt at the same time as the timer reaches 0, the interrupt remains asserted. This action happens because setting the timer interrupt takes precedence over clearing the interrupt. †
2. To clear an active timer interrupt, read the `timer1eoi` register or disable the timer. When the timer is enabled, its interrupt remains asserted until it is cleared by reading the `timer1eoi` register. †

Checking the Interrupt Status

You can query the interrupt status of the timer without clearing its interrupt.

1. To check the interrupt status, read the `timer1intstat` register. †

Masking the Interrupt

The timer interrupt can be masked using the `timer1controlreg` register.

To mask an interrupt, write a 1 to the `timer1_interrupt_mask` bit of the `timer1controlreg` register. †

Timer Address Map and Register Definitions

The address map and register definitions for the HPS-FPGA bridges consist of the following regions:

- OSC1 Timer Module 0
- OSC1 Timer Module 1
- SP Timer Module 0
- SP Timer Module 1

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
For more information, refer to the *Introduction to the Hard Processor System* chapter.
- <https://www.intel.com/content/www/us/en/programmable/hps/arria-v/hps.html>

2021.05.07

av_5v4



Subscribe



Send Feedback

The watchdog timers are peripherals you can use to recover from system lockup that might be caused by software or system related issues. The hard processor system (HPS) provides two programmable watchdog timers, which are connected to the level 4 (L4) peripheral bus. The watchdog timers are instances of the Synopsys DesignWare APB Watchdog Timer (DW_apb_wdt) peripheral. ⁽⁶¹⁾

The microprocessor unit (MPU) subsystem provides two additional watchdog timers.

Related Information

Cortex-A9 Microprocessor Unit Subsystem

For more information about the watchdog timers in the MPU, refer to *Cortex A9 Microprocessor Unit Subsystem* chapter.

Features of the Watchdog Timer

The following list describes the features of the watchdog timer:

- Programmable 32-bit timeout range
- Timer counts down from a preset value to zero, then performs one of the following user-configurable operations:
 - Generates a system reset †
 - Generates an interrupt, restarts the timer, and if the timer is not cleared before a second timeout occurs, generates a system reset
- Dual programmable timeout period, used when the time to wait after the first start is different than that required for subsequent restarts †
- Prevention of accidental restart of the watchdog counter †
- Prevention of accidental disabling of the watchdog counter †
- Pause mode for debugging

⁽⁶¹⁾ Portions © 2017 Synopsys, Inc. Used with permission. All rights reserved. Synopsys & DesignWare are registered trademarks of Synopsys, Inc. All documentation is provided "as is" and without any warranty. Synopsys expressly disclaims any and all warranties, express, implied, or otherwise, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, and any warranties arising out of a course of dealing or usage of trade.

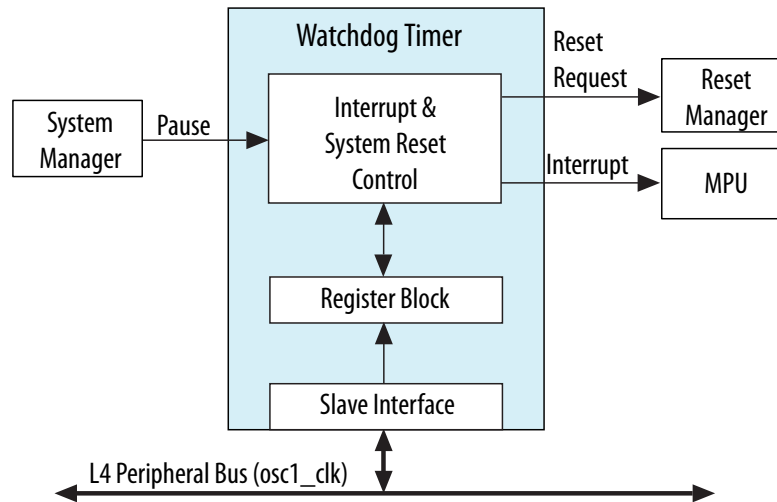
†Paragraphs marked with the dagger (†) symbol are Synopsys Proprietary. Used with permission.

Watchdog Timer Block Diagram and System Integration

Each watchdog timer consists of a slave interface for control and status register (CSR) access, a register block, and a 32-bit down counter that operates on the slave interface clock (`osc1_clk`). A pause input, driven by the system manager, optionally pauses the counter when a CPU is being debugged.

The watchdog timer drives an interrupt request to the MPU and a reset request to the reset manager.

Figure 24-1: Watchdog Timer Block Diagram



Related Information

- [Reset Manager](#) on page 4-1
For more information, refer to the *Reset Manager* chapter.
- [Cortex-A9 Microprocessor Unit Subsystem](#)
For more information about the watchdog timers in the MPU, refer to *Cortex A9 Microprocessor Unit Subsystem* chapter.

Functional Description of the Watchdog Timer

Watchdog Timer Counter

Each watchdog timer is a programmable, little-endian down counter that decrements by one on each clock cycle. The watchdog timer supports 16 fixed timeout period values. Software chooses which timeout periods are desired. A timeout period is $2^{<n>} \text{osc1\_clk}$ clock periods, where n is an integer from 16 to 31 inclusive.

Software must regularly restart the timer (which reloads the counter with the restart timeout period value) to indicate that the system is functioning normally. Software can reload the counter at any time by writing to the restart register. If the counter reaches zero, the watchdog timer has timed out, indicating an unrecoverable error has occurred and a system reset is needed.

Software configures the watchdog timer to one of the following output response modes:

- On timeout, generate a reset request.
- On timeout, assert an interrupt request and restart the watchdog timer. Software must service the interrupt and reset the watchdog timer before a second timeout occurs. Otherwise, generate a reset request.

If a restart occurs at the same time the watchdog counter reaches zero, an interrupt is not generated.

Note: After the watchdog timer reaches zero and generates a reset or interrupt, the counter resets and continues to count.

Related Information

- [Watchdog Timer Clocks](#) on page 24-3
- [Setting the Timeout Period Values](#) on page 24-4
- [Selecting the Output Response Mode](#) on page 24-4
- [Reloading a Watchdog Counter](#) on page 24-5

Watchdog Timer Pause Mode

The watchdog timers can be paused during debugging. The watchdog timer pause mode is controlled by the system manager. The following options are available:

- Pause the timer while either CPU0 or CPU1 is in debug mode
- Pause the timer while only CPU1 is in debug mode
- Pause the timer while only CPU0 is in debug mode
- Do not pause the timer

When pause mode is enabled, the system manager pauses the watchdog timer while debugging. When pause mode is disabled, the watchdog timer runs while debugging.

At reset, the watchdog pausing feature is enabled for both CPUs by default.

Related Information

[Pausing a Watchdog Timer](#) on page 24-5

Watchdog Timer Clocks

Each watchdog timer is connected to the `osc1_clk` clock so that timer operation is not dependent on the phase-locked loops (PLLs) in the clock manager. This independence allows recovery from software that inadvertently programs the PLLs in the clock manager incorrectly.

Related Information

[Clock Manager](#) on page 3-1

For more information, refer to the *Clock Manager* chapter.

Watchdog Timer Resets

Watchdog timers are reset by a cold or warm reset from the reset manager, and are disabled when exiting reset. †

Related Information

[Reset Manager](#) on page 4-1

For more information, refer to the *Reset Manager* chapter.

Taking the Watchdog Timer Out of Reset

When a cold or warm reset is issued in the HPS, the reset manager resets this module and holds it in reset until software releases it.

- Cold reset—HPS I/O are in frozen state (tri-state with a weak pull-up). Pin Mux and GPIO/LoanIO Mux are in the default state.
- Warm reset—HPS I/O retain their configuration. The Pin Mux and GPIO/LoanIO Mux do not change during warm reset, meaning it does not reset to the default/reset value and maintain the current settings. Peripheral IP using the HPS I/O performs proper reset of the signals driving the IO.

After the Cortex-A9 MPCore CPU boots, it can deassert the reset signal by clearing the appropriate bits in the reset manager's corresponding reset register. For details about reset registers, refer to "Module Reset Signals".

FPGA Interface

The watchdog timer interrupts can be routed to the FPGA interface. You can configure and route the interrupts when you instantiate the HPS component in Platform Designer (Standard)

Watchdog Timer Programming Model

Setting the Timeout Period Values

The watchdog timers have a dual timeout period. The counter uses the initial start timeout period value the first the timer is started. All subsequent restarts use the restart timeout period. The valid values are $2^{(16+i)} - 1$ clock cycles, where i is an integer from 0 to 15. To set the programmable timeout periods, perform the following actions in no specific order:

Note: Set the timeout values before enabling the timer.

- To set the initial start timeout period, write i to the timeout period for the initialization field (`top_init`) of the watchdog timeout range register (`wdt_torr`).
- To set the restart timeout period, write i to the timeout period field (`top`) of the `wdt_torr` register

Selecting the Output Response Mode

The watchdog timers have two output response modes. To select the desired mode, perform one of the following actions:

- To generate a system reset request when a timeout occurs, write 0 to the output response mode bit (`rmod`) of the watchdog timer control register (`wdt_cr`).
- To generate an interrupt and restart the timer when a timeout occurs, write 1 to the `rmod` field of the `wdt_cr` register.

If a restart occurs at the same time the watchdog counter reaches zero, a system reset is not generated. †

Related Information

[Watchdog Timer Counter](#) on page 24-2



Enabling and Initially Starting a Watchdog Timer

To enable and start a watchdog timer, write the value 1 to the watchdog timer enable bit (`wdt_en`) of the `wdt_cr` register.

Reloading a Watchdog Counter

To reload a watchdog counter, write the value 0x76 to the counter restart register (`wdt_crr`). This unique 8-bit value is used as a safety feature to prevent accidental restarts.

Pausing a Watchdog Timer

Pausing the watchdog timers is controlled by the L4 watchdog debug register (`wddbq`) in the system manager.

Related Information

[Features of the System Manager](#) on page 6-1

For more information, refer to the *System Manager* chapter.

Disabling and Stopping a Watchdog Timer

The watchdog timers are disabled and stopped by resetting them from the reset manager.

Related Information

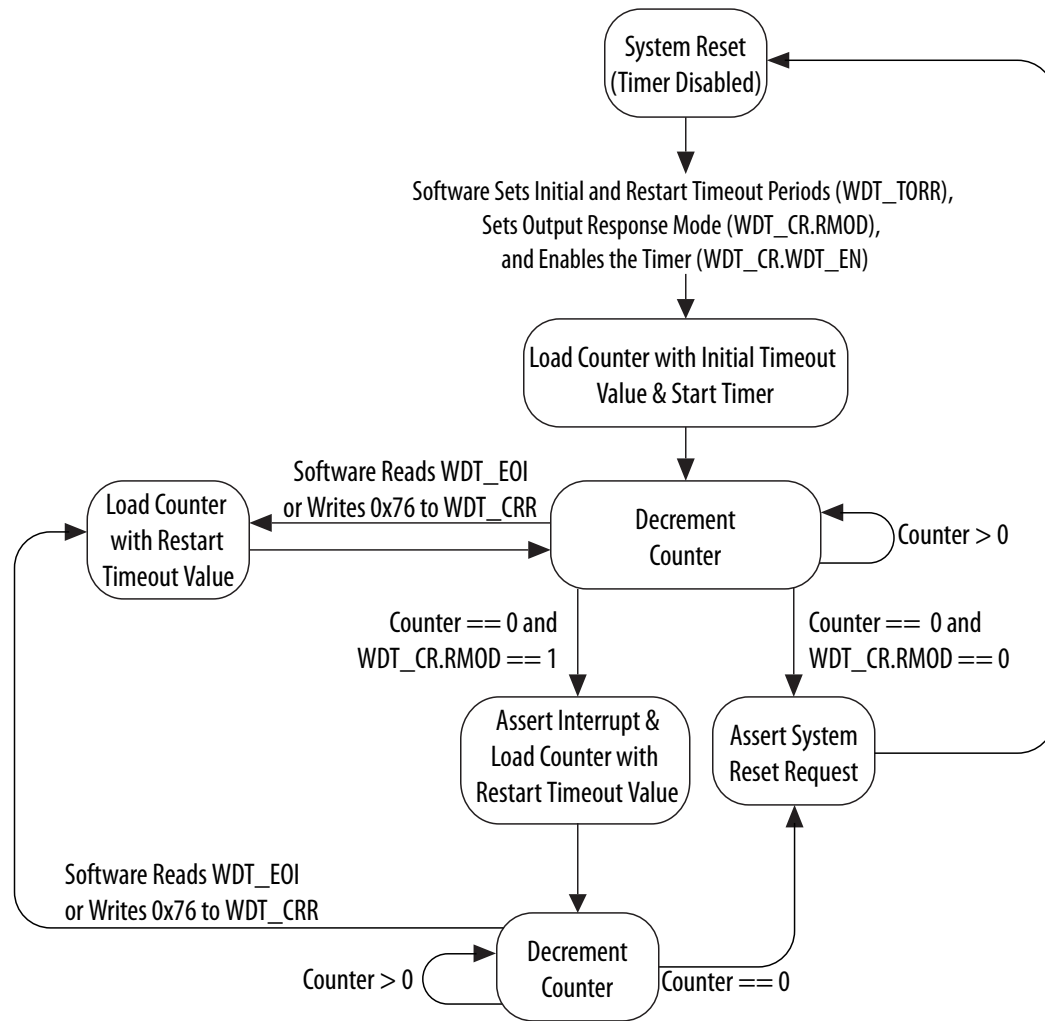
[Reset Manager](#) on page 4-1

For more information, refer to the *Reset Manager* chapter.

Watchdog Timer State Machine

The following figure illustrates the behavior of the watchdog timer, including the behavior of both output response modes. Once initialized, the counter decrements at every clock cycle. The state machine remains in the Decrement Counter state until the counter reaches zero, or the watchdog timer is restarted. If software reads the interrupt clear register (`wdt_eoi`), or writes 0x76 to the `wdt_crr` register, the state changes from Decrement Counter to Load Counter with Restart Timeout Value. In this state, the watchdog counter gets reloaded with the restart timeout value, and then the state changes back to Decrement Counter.

Figure 24-2: Watchdog Timer State Machine



If the counter reaches zero, the state changes based on the value of the output response mode setting defined in the `rmod` bit of the `wdt_cr` register. If the `rmod` bit of the `wdt_cr` register is 0, the output response mode is to generate a system reset request. In this case, the state changes to Assert System Reset Request. In response, the reset manager resets and disables the watchdog timer, and gives software the opportunity to reinitialize the timer.

If the `rmod` bit of the `wdt_cr` register is 1, the output response mode is to generate an interrupt. In this case, the state changes to Assert Interrupt and Load Counter with Restart Timeout Value. An interrupt to the processor is generated, and the watchdog counter is reloaded with the restart timeout value. The state then changes to the second Decrement Counter state, and the counter resumes decrementing. If software reads the `wdt_eoi` register, or writes 0x76 to the `wdt_crr` register, the state changes from Decrement Counter to Load Counter with Restart Timeout Value. In this state, the watchdog counter gets reloaded with the restart timeout value, and then the state changes back to the first Decrement Counter state. If the counter again reaches zero, the state changes to Assert System Reset Request. In response, the reset manager resets the watchdog timer, and gives software the opportunity to reinitialize the timer.

Watchdog Timer Address Map and Register Definitions

The address map and register definitions for the HPS-FPGA bridge consist of the following regions:

- L4 Watchdog Module 0
- L4 Watchdog Module 1

Related Information

- [Introduction to the Hard Processor System](#) on page 2-1
For more information, refer to the *Introduction to the Hard Processor System* chapter.
- <https://www.intel.com/content/www/us/en/programmable/hps/arria-v/hps.html>



Introduction to the HPS Component 25

2021.05.07

av_5v4



Subscribe



Send Feedback

The hard processor system (HPS) component is a wrapper that interfaces logic in the user design to the HPS hard logic, simulation models, BFM, and software handoff files. It instantiates the HPS hard logic in the user design; and enables other soft components to interface with the HPS hard logic. The HPS component itself has a small footprint in the FPGA fabric, because its only purpose is to enable soft logic to connect to the extensive hard logic in the HPS. You can connect soft logic to the HPS.

After the soft logic is connected to the HPS, Platform Designer (Standard) ensures the following features:

- Interoperability by adapting Avalon Memory-Mapped (Avalon-MM) interfaces to AXI
- Handle data width mismatches
- Clock domain transfer crossing

This allows you to integrate IP from Intel, 3rd party IP cores, and custom IP cores to the HPS without having to create integration logic.

For a description of the HPS and its integration into the system on a chip (SoC), refer to the *Cyclone® V Device Datasheet*.

For a description of the HPS system architecture and features, refer to the "Introduction to the Hard Processor" and the CoreSight Debug and Trace chapters in volume 3 of the *Arria V Device Handbook*.

For more information about instantiating the HPS component, refer to the *Instantiating the HPS Component* chapter in the *Hard Processor System Technical Reference Manual*.

For more information about the HPS component interfaces, refer to the *HPS Component Interfaces* chapter in the *Hard Processor System Technical Reference Manual*.

For more information about simulating the HPS component, refer to the *Simulating the HPS Component* chapter in the *Hard Processor System Technical Reference Manual*.

Related Information

- [Intel Arria V Device Datasheet](#)
- [Simulating the HPS Component](#) on page 28-1
- [HPS Component Interfaces](#) on page 27-1
- [Instantiating the HPS Component](#) on page 26-1
- [Introduction to the Hard Processor System](#) on page 2-1
- [CoreSight Debug and Trace](#) on page 10-1

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

MPU Subsystem

The MPU subsystem features the dual Arm Cortex-A9 MPCore processors.

Related Information

[Cortex-A9 Microprocessor Unit Subsystem](#)

Arm CoreSight Debug Components

The following lists the Arm CoreSight debug components:

- Debug Access Port (DAP)
- System Trace Macrocell (STM)
- Trace Funnel
- Embedded Trace FIFO (ETF)
- AMBA Trace Bus Replicator (Replicator)
- Embedded Trace Router (ETR)
- Trace Port Interface Unit (TPIU)
- Embedded Cross Trigger (ECT)
- Program Trace Macrocell (PTM)

Related Information

[CoreSight Debug and Trace](#) on page 10-1

Interconnect

The interconnect consists of the L3 interconnect, SDRAM L3 interconnect, and level 4 (L4) buses. The L3 interconnect is a partially-connected switch fabric.

Related Information

[System Interconnect](#) on page 8-1

HPS-to-FPGA Interfaces

The HPS-to-FPGA interfaces provide a variety of communication channels between the HPS and the FPGA fabric. The HPS is highly integrated with the FPGA fabric, resulting in thousands of connecting signals. Some of the HPS-to-FPGA interfaces include:

- FPGA-to-HPS port
- HPS-to-FPGA port
- Lightweight HPS-to-FPGA port
- FPGA-to-SDRAM interface

Related Information

[HPS-FPGA Bridges](#) on page 9-1

Memory Controllers

The following lists the memory controller peripherals:

- SDRAM L3 Interconnect
- NAND Flash Controller
- Quad SPI Controller
- SD/MMC Controller

Related Information

- [System Interconnect](#) on page 8-1
- [SDRAM Controller Subsystem](#) on page 11-1
- [NAND Flash Controller](#) on page 13-1
- [SD/MMC Controller](#) on page 14-1
- [Quad SPI Flash Controller](#) on page 15-1

Support Peripherals

The following lists the support peripherals:

- Clock Manager
- Reset Manager
- Timers
- Watchdog Timers
- Direct Memory Access (DMA) Controller
- FPGA Manager

Related Information

- [Clock Manager](#) on page 3-1
- [Reset Manager](#) on page 4-1
- [FPGA Manager](#) on page 5-1
- [System Manager](#) on page 6-1
- [DMA Controller](#) on page 16-1
- [Timer](#) on page 23-1
- [Watchdog Timer](#) on page 24-1

Interface Peripherals

The following lists the interface peripherals:

- Ethernet Media Access Controller
- USB 2.0 On-The-Go (OTG) Controllers
- I²C Controllers
- UART

- SPI Master Controllers
- SPI Slave Controllers
- GPIO Interfaces

Related Information

- [Ethernet Media Access Controller](#) on page 17-1
- [USB 2.0 OTG Controller](#) on page 18-1
- [SPI Controller](#) on page 19-1
- [I2C Controller](#) on page 20-1
- [UART Controller](#) on page 21-1
- [General-Purpose I/O Interface](#) on page 22-1

On-Chip Memories

The following lists the on-chip memories:

- On-Chip RAM
- Boot ROM

Related Information

[On-Chip Memory](#) on page 12-1

Instantiating the HPS Component 26

2021.05.07

av_5v4



Subscribe



Send Feedback

You instantiate the hard processor system (HPS) component in Platform Designer (Standard). The HPS is available in the Platform Designer (Standard) IP catalog under **Processor and Peripherals > Hard Processor Systems**. This chapter describes the parameters available in the HPS component parameter editor, which opens when you add or edit an HPS component.

Related Information

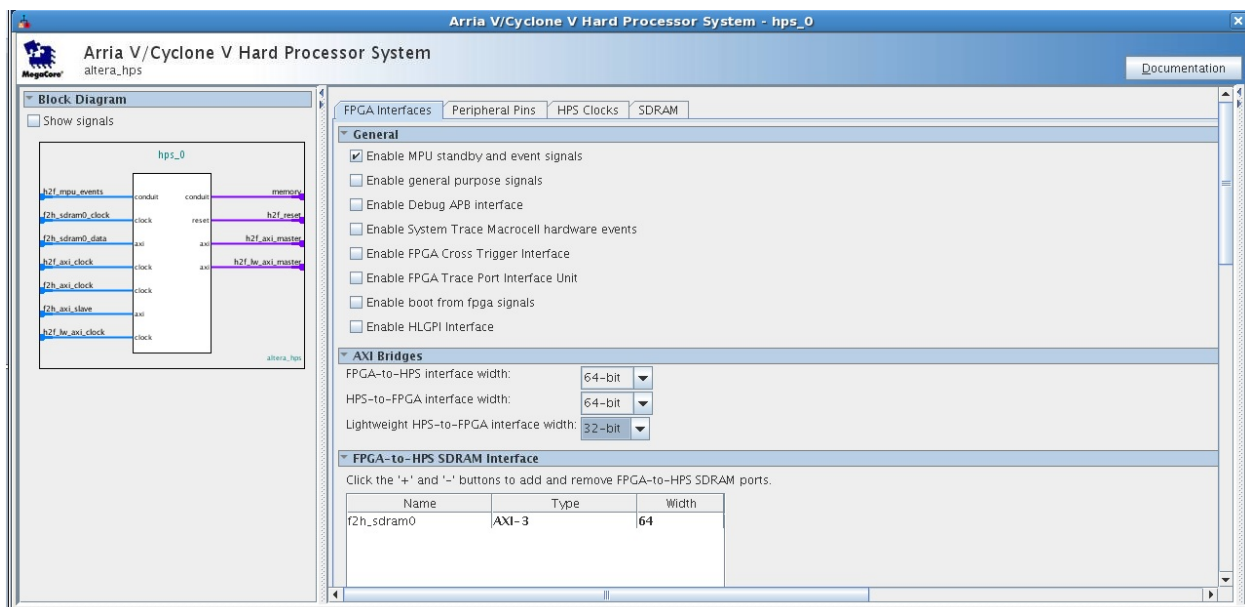
- [Intel Arria V Device Datasheet](#)
- [Intel® Quartus® Prime Standard Edition Handbook Volume 1: Design and Synthesis](#)

For general information about using Platform Designer (Standard), refer to the *Creating a System with Platform Designer (Standard)* chapter in the *Quartus® Prime Handbook*.

FPGA Interfaces

The **FPGA Interfaces** tab is one of four tabs on the HPS component. This tab contains several groups with the following parameters:

Figure 26-1: FPGA Interface Tab



Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

Related Information[Introduction to the HPS Component](#) on page 25-1

General Interfaces

When enabled, the interfaces described in the following table become visible in the HPS component.

Table 26-1: General Parameters

Parameter Name	Parameter Description	Interface Name
Enable MPU standby and event signals	Enables interfaces that perform the following functions: - Notify the FPGA fabric that the microprocessor unit (MPU) is in standby mode. "16">Wake up an MPCore processor from a wait for event (WFE) state.	h2f_mpu_events
Enable general purpose signals	Enables a pair of 32-bit unidirectional general-purpose interfaces between the FPGA fabric and the FPGA manager in the HPS portion of the SoC device.	h2f_gp
Enable Debug APB interface	Enables debug interface to the FPGA, allowing access to debug components in the HPS. For more information, refer to the <i>CoreSight Debug and Trace</i> chapter.	h2f_debug_apb h2f_debug_apb_sideband h2f_debug_apb_clock h2f_debug_apb_reset
Enable System Trace Macrocell hardware events	Enables system trace macrocell (STM) hardware events, allowing logic inside the FPGA to insert messages into the trace stream. For more information, refer to the <i>CoreSight Debug and Trace</i> chapter.	f2h_stm_hw_events
Enable FPGA Cross Trigger interface	Enables the cross trigger interface (CTI), which allows trigger sources and sinks to interface with the embedded cross trigger (ECT). For more information, refer to the <i>CoreSight Debug and Trace</i> chapter.	h2f_cti h2f_cti_clock

Parameter Name	Parameter Description	Interface Name
Enable FPGA Trace Port Interface Unit	Enables an interface between the trace port interface unit (TPIU) and logic in the FPGA. The TPIU is a bridge between on-chip trace sources and a trace port. For more information, refer to the <i>CoreSight Debug and Trace</i> chapter.	h2f_tpiu h2f_tpiu_clock_in h2f_tpiu_clock
Enable boot from FPGA signals	Enables an input to the HPS indicating whether a preloader is available in the on-chip memory of the FPGA. This option also enables a separate input to the HPS indicating a fallback preloader is available in the FPGA memory. A fallback preloader is used when there is no valid preloader image found in flash memory. For more information, refer to <i>Appendix A: Booting and Configuration</i> .	f2h_boot_from_fpga
Enable HLGPI Interface	Enables a general purpose interface that is connected to the General Purpose I/O (GPIO) peripheral of HPS. This is an input-only interface with 14-bit width. This interface shares the I/O pins with the HPS DDR SDRAM controller.	hps_io (hps_io_gpio_inst_HLGPI[0..13])

Related Information

- [CoreSight Debug and Trace](#) on page 10-1
Enabling the TPIU exposes trace signals to the device pins. Refer to the *CoreSight Debug and Trace* for more information.
- [Booting and Configuration](#) on page 29-1
For detailed information about the HPS boot sequence, refer to the *Booting and Configuration*.

FPGA-to-HPS SDRAM Interface

In the **FPGA-to-HPS SDRAM Interface** table, use + or – to add or remove FPGA-to-HPS SDRAM interfaces.

You can add one or more SDRAM ports that make the HPS SDRAM subsystem accessible to the FPGA fabric.

Table 26-2: FPGA-to-HPS SDRAM Port and Interface Names

Port Name	Interface Name
f2h_sdram0	f2h_sdram0_data f2h_sdram0_clock
f2h_sdram1	f2h_sdram1_data f2h_sdram1_clock
f2h_sdram2	f2h_sdram2_data f2h_sdram2_clock
f2h_sdram3	f2h_sdram3_data f2h_sdram3_clock
f2h_sdram4	f2h_sdram4_data f2h_sdram4_clock
f2h_sdram5	f2h_sdram5_data f2h_sdram5_clock

The following table shows the parameters available for each SDRAM interface where the **Name** parameter denotes the interface name.

Table 26-3: FPGA-to-HPS SDRAM Interface Parameters

Parameter Name	Parameter Description
Name	Port name (auto assigned as shown in Table 26-2 table below)
Type	Interface type: • AXI-3 • Avalon-MM Bidirectional • Avalon-MM Write-only • Avalon-MM Read-only
Width	32, 64, 128, or 256

Note: You can configure the slave interface to a data width of 32, 64, 128, or 256 bits. To facilitate accessing this slave from a memory-mapped master with a smaller address width, you can use the *Intel Address Span Extender*.

Related Information

- [Using the Address Span Extender Component](#) on page 26-15
For information about Address sfo1410144425160 extender
- [SDRAM Controller Subsystem](#) on page 11-1

DMA Peripheral Request

You can enable each direct memory access (DMA) controller peripheral request ID individually. Each request ID enables an interface for FPGA soft logic to request one of eight logical DMA channels to the FPGA.

Related Information

[DMA Controller](#) on page 16-1

For more information, refer to the *DMA Controller* chapter in the *Arria V Device Handbook, Volume 3*.

Interrupts

Table 26-4: FPGA-to-HPS Interrupts Interface

Parameter Name	Parameter Description	Interface Name
Enable FPGA-to-HPS Interrupts	Enables the interface for FPGA interrupt signals to the MPU (in the HPS).	f2h_irq0 f2h_irq1

You can enable the interfaces for each HPS peripheral interrupt to the FPGA.

Table 26-5: HPS-to-FPGA Interrupts Interface

The following table lists the available HPS-to-FPGA interrupt interfaces and the corresponding parameters to enable them.

Parameter Name	Parameter Description	Interface Name
Enable clock peripheral interrupts	Enables the interface for HPS clock manager and MPU wake-up interrupt signals to the FPGA	h2f_clkmgr_interrupt h2f_mpuwakeupt_interrupt
Enable CTI interrupts	Enables the interface for HPS cross-trigger interrupt signals to the FPGA	h2f_cti_interrupt0 h2f_cti_interrupt1

Parameter Name	Parameter Description	Interface Name
Enable DMA interrupts	Enables the interface for HPS DMA channels interrupt and DMA abort interrupt to the FPGA	h2f_dma_interrupt0 h2f_dma_interrupt1 h2f_dma_interrupt2 h2f_dma_interrupt3 h2f_dma_interrupt4 h2f_dma_interrupt5 h2f_dma_interrupt6 h2f_dma_interrupt7 h2f_dma_abort_interrupt
Enable EMAC interrupts	Enables the interface for HPS Ethernet MAC controller interrupt to the FPGA	h2f_emac0_interrupt h2f_emac1_interrupt
Enable FPGA manager interrupts	Enables the interface for the HPS FPGA manager interrupt to the FPGA	h2f_fpga_man_interrupt
Enable GPIO interrupts	Enables the interface for the HPS general purpose IO (GPIO) interrupt to the FPGA	h2f_gpio0_interrupt h2f_gpio1_interrupt h2f_gpio2_interrupt
Enable I ² C-EMAC interrupts (for I2C2 and I2C3)	Enables the interface for the HPS I ² C interrupt to the FPGA (I ² C used for EMAC media interface control)	h2f_i2c_emac0_interrupt h2f_i2c_emac1_interrupt
Enable I ² C peripheral interrupts (for I2C0 and I2C1)	Enables the interface for the HPS I ² C interrupt to the FPGA (I ² C used for general purpose)	h2f_i2c0 _interrupt h2f_i2c1_interrupt
Enable L4 timer interrupts	Enables interface for the HPS SP timer interrupt to the FPGA. For more information, refer to the Timer Clock Characteristics table in the <i>Timer</i> chapter in the <i>Arria V Device Handbook, Volume 3</i> .	h2f_l4sp0_interrupt h2f_l4sp1_interrupt
Enable NAND interrupts	Enables the interface for the HPS NAND controller interrupt to the FPGA	h2f_nand_interrupt

Parameter Name	Parameter Description	Interface Name
Enable OSC timer interrupts	Enables the interface for the HPS OSC timer interrupt to the FPGA. For more information, refer to the Timer Clock Characteristics table in the <i>Timer</i> chapter in the <i>Arria V Device Handbook, Volume 3</i> .	h2f_osc0_interrupt h2f_osc1_interrupt
Enable Quad SPI interrupts	Enables interface for the HPS QSPI controller interrupt to the FPGA	h2f_qspi_interrupt
Enable SD/MMC interrupts	Enables the interface for the HPS SD/MMC controller interrupt to the FPGA	h2f_sdmmc_interrupt
Enable SPI master interrupts	Enables the interface for the HPS SPI master controller interrupt to the FPGA	h2f_spi0_interrupt h2f_spi1_interrupt
Enable SPI slave interrupts	Enables the interface for the HPS SPI slave controller interrupt to the FPGA	h2f_spi2_interrupt h2f_spi3_interrupt
Enable UART interrupts	Enables the interface for the HPS UART controller interrupt to the FPGA	h2f_uart0_interrupt h2f_uart1_interrupt
Enable USB interrupts	Enables the interface for the HPS USB controller interrupt to the FPGA	h2f_usb0_interrupt h2f_usb1_interrupt
Enable watchdog interrupts	Enables the interface for the HPS watchdog interrupt to the FPGA	h2f_wdog0_interrupt h2f_wdog1_interrupt

AXI Bridges

Table 26-6: Bridge Parameters

Parameter Name	Parameter Description	Interface Name
FPGA-to-HPS interface width	Enable or disable the FPGA-to-HPS interface; if enabled, set the data width to 32, 64, or 128 bits.	f2h_axi_slave f2h_axi_clock

Parameter Name	Parameter Description	Interface Name
HPS-to-FPGA interface width	Enable or disable the HPS-to-FPGA interface; if enabled, set the data width to 32, 64, or 128 bits.	h2f_axi_master h2f_axi_clock
Lightweight HPS-to-FPGA interface width	Enable or disable the lightweight HPS-to-FPGA interface. When enabled, the data width is 32 bits.	h2f_lw_axi_master h2f_lw_axi_clock

Note: To facilitate accessing these slaves from a memory-mapped master with a smaller address width, you can use the Intel Platform Designer (Standard) Address Span Extender.

Related Information

[Using the Address Span Extender Component](#) on page 26-15

For information about Address sfo1410144425160 extender

Configuring HPS Clocks and Resets

The **HPS Clocks** tab is one of four tabs on the HPS Component. This tab contains several groups with the following parameters:

[User Clocks](#) on page 26-8

[Reset Interfaces](#) on page 26-9

[PLL Reference Clocks](#) on page 26-10

[Peripheral FPGA Clocks](#) on page 26-11

User Clocks

When you enable a HPS-to-FPGA user clock, you must manually enter its maximum frequency for timing analysis. The Timing Analyzer has no other information about how software running on the HPS configures the phase-locked loop (PLL) outputs. Each possible clock, including clocks that are available from peripherals, has its own parameter for describing the clock frequency.

Note: Use the `set_global_assignment -name ENABLE_HPS_INTERNAL_TIMING ON` to enable timing analysis.

Related Information

[Selecting PLL Output Frequency and Phase](#) on page 26-15

User Clock Parameters

The frequencies that you provide are the maximum expected frequencies. The actual clock frequencies can be modified through the register interface, for example by software running on the microprocessor unit (MPU). For further details, refer to *Selecting PLL Output Frequency and Phase*.

Table 26-7: User Clock Parameters

Parameter Name	Parameter Description	Clock Interface Name
Enable HPS-to-FPGA user 0 clock	Enable main PLL from HPS-to-FPGA	h2f_user0_clock
User 0 clock frequency	Specify the maximum expected frequency for the main PLL	
Enable HPS-to-FPGA user 1 clock	Enable peripheral PLL from HPS-to-FPGA	h2f_user1_clock
User 1 clock frequency	Specify the maximum expected frequency for the peripheral PLL	
Enable HPS-to-FPGA user 2 clock	Enable SDRAM PLL from HPS-to-FPGA	h2f_user2_clock
User 2 clock frequency	Specify the maximum expected frequency for the SDRAM PLL	

Related Information

[Selecting PLL Output Frequency and Phase](#) on page 26-15

Clock Frequency Usage

The clock frequencies you provide are reported in a Synopsys Design Constraints File (.sdc) generated by Platform Designer (Standard). The .sdc file is referenced in the system .qip file when the system is generated.

Related Information

- [Selecting PLL Output Frequency and Phase](#) on page 26-15
- [Clock Manager](#) on page 3-1
For general information about clock signals, refer to the *Clock Manager* chapter in the *Arria V Device Handbook, Volume 3*.

Reset Interfaces

You can enable the resets on an individual basis through the **FPGA Interfaces** tab under the **Resets** section.

Table 26-8: Reset Parameters

Parameter Name	Parameter Description	Interface Name
Enable HPS-to-FPGA cold reset output	Enable interface for HPS-to-FPGA cold reset output	h2f_cold_reset

Parameter Name	Parameter Description	Interface Name
Enable HPS warm reset handshake signals	Enable an additional pair of reset handshake signals allowing soft logic to notify the HPS when it is safe to initiate a warm reset in the FPGA fabric.	h2f_warm_reset_handshake
Enable FPGA-to-HPS debug reset request	Enable interface for FPGA-to-HPS debug reset request	f2h_debug_reset_req
Enable FPGA-to-HPS warm reset request	Enable interface for FPGA-to-HPS warm reset request	f2h_warm_reset_req
Enable FPGA-to-HPS cold reset request	Enable interface for FPGA-to-HPS cold reset request	f2h_cold_reset_req

Related Information

[Reset Manager](#) on page 4-1

For more information about the reset interfaces, refer to *Functional Description of the Reset Manager* in the *Reset Manager* chapter in the *Arria V Device Handbook, Volume 3*.

PLL Reference Clocks

Table 26-9: PLL Reference Clock Parameters

Parameter Name	Parameter Description	Clock Interface Name
Enable FPGA-to-HPS peripheral PLL reference clock	Enable the interface for FPGA fabric to supply reference clock to HPS peripheral PLL	f2h_periph_ref_clock
Enable FPGA-to-HPS SDRAM PLL reference clock	Enable the interface for FPGA fabric to supply reference clock to HPS SDRAM PLL	f2h_sdram_ref_clock

Related Information

[Clock Manager](#) on page 3-1

For general information about clock signals, refer to the *Clock Manager* chapter in the *Arria V Device Handbook, Volume 3*.

Peripheral FPGA Clocks

Table 26-10: Peripheral FPGA Clock Parameters

Parameter Name	Parameter Description
EMAC 0 (emac0_md_clk clock frequency)	If EMAC 0 peripheral is routed to FPGA, use the input field to specify EMAC 0 MDIO clock frequency
EMAC 0 (emac0_gtx_clk clock frequency)	If EMAC 0 peripheral is routed to FPGA, use the input field to specify EMAC 0 transmit clock frequency
EMAC 1 (emac1_md_clk clock frequency)	If EMAC 1 peripheral is routed to FPGA, use the input field to specify EMAC 1 MDIO clock frequency
EMAC 1 (emac1_gtx_clk clock frequency)	If EMAC 1 peripheral is routed to FPGA, use the input field to specify EMAC 1 transmit clock frequency
QSPI (qspi_sclk_out clock frequency)	If QSPI peripheral is routed to FPGA, use the input field to specify QSPI serial clock frequency
SPIM 0 (spim0_sclk_out clock frequency)	If SPI master 0 peripheral is routed to FPGA, use the input field to specify SPI master 0 output clock frequency
SPIM 1 (spim1_sclk_out clock frequency)	If SPI master 1 peripheral is routed to FPGA, use the input field to specify SPI master 1 output clock frequency
I ² C0 (i2c0_clk clock frequency)	If I ² C 0 peripheral is routed to FPGA, use the input field to specify I ² C 0 output clock frequency
I ² C1 (i2c1_clk clock frequency)	If I ² C 1 peripheral is routed to FPGA, use the input field to specify I ² C 1 output clock frequency
I ² C2 (i2c2_clk clock frequency)	If I ² C 2 peripheral is routed to FPGA, use the input field to specify I ² C 2 output clock frequency
I ² C3 (i2c3_clk clock frequency)	If I ² C 3 peripheral is routed to FPGA, use the input field to specify I ² C 3 output clock frequency

Configuring Peripheral Pin Multiplexing

The **Peripheral Pin Multiplexing** tab is one of four tabs on the HPS Component. This tab contains several groups with the following parameters:

[Configuring Peripherals](#) on page 26-12

[Connecting Unassigned Pins to GPIO](#) on page 26-12

[Using Unassigned IO as LoanIO](#) on page 26-13

You can utilize unused HPS IOs as LoanIO, which is directly driven by the FPGA and can be used as input, output, or bi-directional IO.

[Resolving Pin Multiplexing Conflicts](#) on page 26-13

Use the **Peripherals MUX Table** to view pins with invalid multiple assignments.

[Peripheral Signals Routed to FPGA](#) on page 26-14

Configuring Peripherals

The **Peripheral Pin Multiplexing** tab contains a group of parameters for each available type of peripheral. You can enable one or more instances of each peripheral type by selecting an HPS I/O pin set for each instance. When enabled, some peripherals also have mode settings specific to their functions.

When you assign a peripheral to an HPS I/O pin set, the corresponding peripheral is highlighted in the **Peripherals MUX Table** at the end of **Peripheral Pin Multiplexing** tab. The other unused peripheral pin set remains un-highlighted in the table.

Each list in the **Peripheral Pin Multiplexing** tab has a hint describing in detail the options available in the list. The hint for each mode list shows the signals used by each available mode. View each hint by hovering over the corresponding mode list.

You can enable the following types of peripherals. For details of peripheral-specific settings, refer to the chapter for each peripheral:

Related Information

- [CoreSight Debug and Trace](#) on page 10-1
Enabling the TPIU exposes trace signals to the device pins. Refer to the *CoreSight Debug and Trace* for more information.
- [NAND Flash Controller](#) on page 13-1
- [SD/MMC Controller](#) on page 14-1
Secure Digital / MultiMediaCard (SD/MMC) Controller chapter in the *Arria V Device Handbook, Volume 3*.
- [Quad SPI Flash Controller](#) on page 15-1
Quad serial peripheral interface (SPI) Flash Controller chapter in the *Arria V Device Handbook, Volume 3*.
- [Ethernet Media Access Controller](#) on page 17-1
Ethernet Media Access Controller chapter in the *Arria V Device Handbook, Volume 3*.
- [USB 2.0 OTG Controller](#) on page 18-1
- [SPI Controller](#) on page 19-1
- [I2C Controller](#) on page 20-1
- [UART Controller](#) on page 21-1

Connecting Unassigned Pins to GPIO

For pins that are not assigned to any peripherals, you can assign them to the GPIO peripheral by clicking on the corresponding GPIO push button in the table. Click on the selected push button to remove GPIO pin assignment. The table also shows the GPIO bit number that correlates to the I/O pins.

Using Unassigned IO as LoanIO

You can utilize unused HPS IOs as LoanIO, which is directly driven by the FPGA and can be used as input, output, or bi-directional IO.

Each LoanIO port has an input, output, and output enable, which directly controls the HPS IO functions. The LoanIO only operates when the HPS registers have been set up in the pre-loader to allow their operation. The LoanIO are asynchronous, thus no clocking is required.

The status of the LoanIO in the duration after HPS I/O is configured and before the FPGA is configured is as follows: When the FPGA powers up and is "not" configured, inputs into the HPS from the FPGA are driven to a logical 1. When the FPGA is configured the signal may toggle, and then takes on whatever level the FPGA user design drives out.

Use the following steps to enable the LoanIO signals:

1. Select the **Peripheral Pins Multiplexing** tab in the HPS parameter editor.
2. Choose the corresponding LoanIO pins from the **Peripherals Mux Table**, and click the push button to select/unselect it.
3. Export the peripheral signals out of the Platform Designer (Standard) system.
4. In the Quartus Prime software, connect the user logic to the LoanIO interface to drive the HPS IOs.

Table 26-11: Generated Conduit Signal Interface

Conduit Name	Direction	Declarations
._hps_io_gpio_inst_LOANIOXX	Bi-direction	User must declare as a top-level pin; pin assignment is hardcoded following the HPS IO location.
._h2f_loan_io_in	Out	HPS IO data input signal, output to FPGA user logic.
._h2f_loan_io_out	In	HPS IO data output signal, input from FPGA user logic.
._h2f_loan_io_oe	In	HPS IO data output enable signal, input from FPGA user logic.

Platform Designer (Standard) generates a full signal array for h2f_loan_io_in, h2f_loan_io_out, and h2f_loan_io_oe. You must assign user logic to the specific signal array. For example, you have triggered LoanIO 40, so its respective signal array is h2f_loan_io_in[40], h2f_loan_io_out[40], and h2f_loan_io_oe[40].

Resolving Pin Multiplexing Conflicts

Use the **Peripherals MUX Table** to view pins with invalid multiple assignments.

Pins that have multiple peripherals assigned to them are highlighted in red color with the conflicting peripherals in boldface font style. Solve the multiplexing conflicts by de-selecting peripherals that are assigned to the pins, leaving only one peripheral, which is needed.

Peripheral Signals Routed to FPGA

You can route the peripheral signals to the FPGA fabric and assign them to the FPGA I/O pins.

The following steps show you how to enable the peripheral signals:

1. Select FPGA in the peripheral pin multiplexing selection drop-down box.
2. Export the peripheral signals out of Platform Designer (Standard) system.
3. In the Intel Quartus Prime software, connect the signals to the FPGA I/O pins.

Note: When routed to the FPGA, some HPS peripherals require additional pipeline support in the connected soft logic. Routing into the FPGA circumvents the pipelining available in the HPS I/O. Refer to the relevant HPS peripheral chapter for details.

Configuring the External Memory Interface

The **SDRAM** tab is one of four tabs on the HPS component. This tab contains the PLL output frequency and phase group.

The HPS supports one memory interface implementing double data rate 2 (DDR2), double data rate 3 (DDR3), and low-power double data rate 2 (LPDDR2) protocols. The interface can be up to 40 bits wide with optional error correction code (ECC).

Configuring the HPS SDRAM controller is similar to configuring any other SDRAM controller in Platform Designer (Standard). There are several important differences:

- The HPS parameter editor supports all SDRAM protocols with one tab. When you parameterize the SDRAM controller, you must specify the memory protocol: DDR2, DDR3, or LPDDR2.

To select the memory protocol, select DDR2, DDR3, or LPDDR2 from the **SDRAM Protocol** list in the **SDRAM** tab. After you select the protocol, settings not applicable to that protocol are disabled.

- Many HPS SDRAM controller settings are the same as for Intel dedicated DDR2, DDR3, and LPDDR2 controllers. This section only describes SDRAM parameters that are specific to the HPS component.
- Because the HPS memory controller is not configurable through the Quartus Prime software, the Controller and Diagnostic tabs are not present in the HPS parameter editor.
- Some settings, such as the controller settings, are not included because they can only be configured through the register interface, for example by software running on the MPU.
- Unlike the memory interface clocks in the FPGA, the memory interface clocks for the HPS are initialized by the boot-up code using values provided by the configuration process. You can accept the values provided by UniPHY, or you can use your own PLL settings, as described in *Selecting PLL Output Frequency and Phase*.

Note: The HPS does not support external memory interface (EMIF) synthesis generation, compilation, or timing analysis.

The HPS memory controller cannot be bonded with a memory controller on the FPGA portion of the device.

For detailed information about SDRAM controller parameters, refer to the following chapters:

Related Information

- [Selecting PLL Output Frequency and Phase](#) on page 26-15
- [External Memory Interface Handbook Volume 2: Design Guidelines](#)
The Implementing and Parameterizing Memory IP chapter in the External Memory Interface Handbook.

- [External Memory Interface Handbook Volume 3: Reference Material](#)
The Functional Description--Hard Memory Interface chapter in the *External Memory Interface Handbook*. "EMI-Related HPS Features in SoC Devices" describes features specific to the HPS SDRAM controller.
- [SDRAM Controller Subsystem](#) on page 11-1

Selecting PLL Output Frequency and Phase

You select PLL output frequency and phase with controls in the **PHY Settings** tab. In the HPS, PLL frequencies and phases are set by software at system startup. A PLL might not be able to produce the exact frequency that you specify in **Memory clock frequency**. Normally, the Quartus Prime software sets **Achieved memory clock frequency** to the closest achievable frequency, using an algorithm that tries to balance frequency accuracy against clock jitter. This clock frequency is used for timing analysis by the Timing Analyzer.

It is possible to use a different software algorithm for configuring the PLLs. You can force the **Achieved memory clock frequency** box to take on the same value as **Memory clock frequency**, by turning on **Use specified frequency instead of calculated frequency** in the **PHY Settings** tab, under **Clocks**.

Note: If you turn on **Use specified frequency instead of calculated frequency**, the Quartus Prime software assumes that the value in the **Achieved memory clock frequency** box is correct. If it is not, timing analysis results are incorrect.

Related Information

- [External Memory Interface Handbook Volume 2: Design Guidelines](#)
The Implementing and Parameterizing Memory IP chapter in the *External Memory Interface Handbook*.
- [External Memory Interface Handbook Volume 3: Reference Material](#)
The Functional Description--Hard Memory Interface chapter in the *External Memory Interface Handbook*. "EMI-Related HPS Features in SoC Devices" describes features specific to the HPS SDRAM controller.

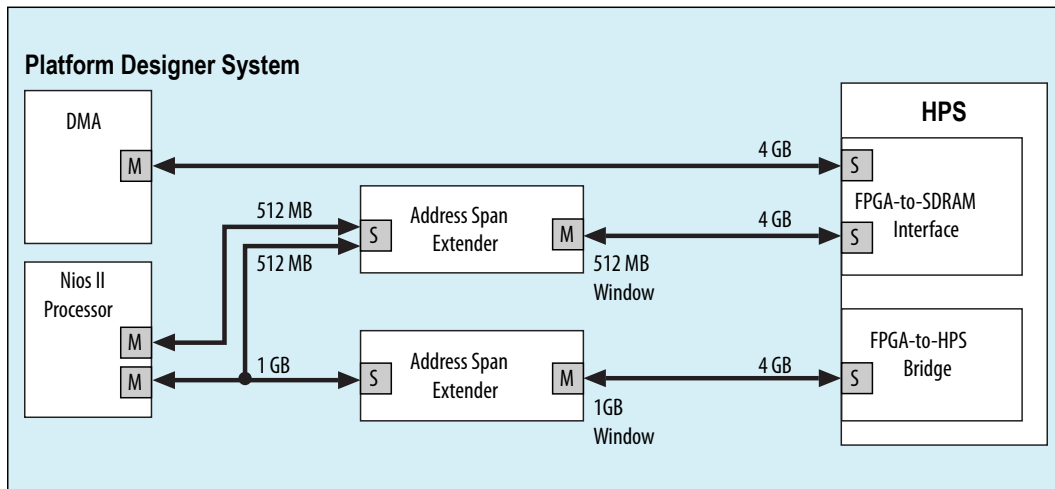
Using the Address Span Extender Component

The FPGA-to-HPS bridge and FPGA-to-HPS SDRAM memory-mapped interfaces expose their entire 4-GB address spaces to the FPGA fabric. The address span extender component provides a memory-mapped window into the address space that it masters. Using the address span extender, you can expose portions of the HPS memory space without needing to expose the entire 4 GB address space.

You can use the address span extender between a soft logic master and an FPGA-to-HPS bridge or FPGA-to-HPS SDRAM interface. This component reduces the number of address bits required for a master to address a memory-mapped slave interface located in the HPS.

Figure 26-2: Address Span Extender Components

Two address span extender components used in a system with the HPS.



You can also use the address span extender in the HPS-to-FPGA direction, for slave interfaces in the FPGA. In this case, the HPS-to-FPGA bridge exposes a limited, variable address space in the FPGA, which can be paged in using the address span extender.

For example, suppose that the HPS-to-FPGA bridge has a 1-GB span, and the HPS needs to access three independent 1-GB memories in the FPGA portion of the device. To achieve this, the HPS programs the address span extender to access one SDRAM (1-GB) in the FPGA at a time. This technique is commonly called paging or windowing.

Related Information

- **Intel Quartus Prime Standard Edition User Guide: Platform Designer**
For more information about the Address Span Extender, refer to the Address Span Extender section.
- **Platform Designer (Standard) Interconnect and System Design Components**
For more information about the address span extender, refer to *Bridges* in the *Platform Designer (Standard) Interconnect and System Design Components* chapter in the *Quartus® Prime Handbook*.

Generating and Compiling the HPS Component

The process of generating and compiling an HPS design is very similar to the process for any other Platform Designer (Standard) project. Perform the following steps:

1. Generate the design with Platform Designer (Standard). The generated files include an **.sdc** file containing clock timing constraints. If simulation is enabled, simulation files are also generated.
2. Add `<system_name>.qip` to the Quartus Prime project. `<qsys_system_name>.qip` is the Quartus Prime IP file for the HPS component, generated by Platform Designer (Standard).
3. Perform analysis and synthesis with the Quartus Prime software.
4. Assign constraints to the SDRAM component. When Platform Designer (Standard) generates the HPS component (step 1), it generates the pin assignment Tcl Script File (**.tcl**) to perform memory assignments. The script file name is `<system_name>_pin_assignments.tcl`, where

<system_name> is the name of your Platform Designer (Standard) system. Run this script to assign constraints to the SDRAM component.

Note: For information about running the pin assignment script, refer to “IP catalog” in the *Implementing and Parameterizing Memory IP* chapter in the *External Memory Interface Handbook*.

You do not need to specify pin assignments other than memory assignments. When you configure pin multiplexing as described in *Configuring Peripheral Pin Multiplexing* section, you implicitly make pin assignments for all HPS peripherals. Each peripheral is routed exclusively to the pins you specify. HPS I/O signals are exported to the top level of the Platform Designer (Standard) design, with information enabling the Quartus Prime software to make pin assignments automatically.

You can view and modify the assignments in the **Peripheral Pin Multiplexing** tab. You can also view the assignments in the Quartus fitter report.

5. Compile the design with the Quartus Prime software.
6. Optionally back-annotate the SDRAM pin assignments, to eliminate pin assignment warnings the next time you compile the design.

Related Information

- [Configuring Peripheral Pin Multiplexing](#) on page 26-11
- [Intel® Quartus® Prime Standard Edition Handbook Volume 1: Design and Synthesis](#)
For general information about using Platform Designer (Standard), refer to the *Creating a System with Platform Designer (Standard)* chapter in the *Quartus® Prime Handbook*.
- [External Memory Interface Handbook Volume 2: Design Guidelines](#)
The Implementing and Parameterizing Memory IP chapter in the *External Memory Interface Handbook*.
- [Setting Up the HPS Component for Simulation](#) on page 28-3
For a description of the simulation files generated, refer to "Simulation Flow" in the *Simulating the HPS Component* chapter of the *Arria V Device Handbook, Volume 3*.

2021.05.07

av_5v4



Subscribe



Send Feedback

This chapter describes the interfaces, including clocks and resets, implemented by the hard processor system (HPS) component.

The majority of the resets can be enabled on an individual basis. The exception is the `h2f_reset` interface, which is always enabled.

You must declare the clock frequency of each HPS-to-FPGA clock for timing purposes. Each possible clock, including ones that are available from peripherals, has its own parameter for describing the clock frequency. Declaring the clock frequency for HPS-to-FPGA clocks specifies how you plan to configure the PLLs and peripherals, to enable the Timing Analyzer to accurately estimate system timing. It has no effect on PLL settings.

Related Information

- [Avalon Interface Specifications](#)

For Avalon protocol timing, refer to *Avalon Interface Specifications*.

- [HPS Component Interfaces](#) on page 27-1

For information about instantiating the HPS component, refer to the *Instantiating the HPS Component* chapter.

Memory-Mapped Interfaces

FPGA-to-HPS Bridge

Table 27-1: FPGA-to-HPS Bridges and Clocks

Interface Name	Description	Associated Clock Interface
<code>f2h_axi_slave</code>	FPGA-to-HPS AXI slave interface	<code>f2h_axi_clock</code>

The FPGA-to-HPS interface is a configurable data width AXI slave allowing FPGA masters to issue transactions to the HPS. This interface allows the FPGA fabric to access the majority of the HPS slaves. This interface also provides a coherent memory interface.

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

The FPGA-to-HPS interface is an AXI-3 compliant interface with the following features:

- Configurable data width: 32, 64, or 128 bits
- Accelerator Coherency Port (ACP) sideband signals
- HPS-side AXI bridge to manage clock crossing, buffering, and data width conversion

Other interface standards in the FPGA fabric, such as connecting to Avalon® Memory-Mapped (Avalon-MM) interfaces, can be supported through the use of soft logic adapters. The Platform Designer (Standard) system integration tool automatically generates adapter logic to connect AXI to Avalon-MM interfaces.

This interface has an address width of 32 bits. To access existing Avalon-MM/AXI masters, you can use the Intel Address Span Extender.

Related Information

- [Clocks](#) on page 27-4
- [Features of the HPS-FPGA Bridges](#) on page 9-1
For more information, refer to the *HPS FPGA AXI Bridges Component* chapter.
- [HPS Component Interfaces](#) on page 27-1
For information about the address span extender, refer to “Using the Address Span Extender Component” in the *Instantiating the HPS Component* chapter.

ACP Sideband Signals

For communication with the ACP on the microprocessor unit (MPU) subsystem, AXI sideband signals are used to describe the inner cacheable attributes for the transaction.

Related Information

[Cortex-A9 MPCore](#)

For more information about the ACP sideband signals, refer to the *Cortex-A9 MPU Subsystem* chapter.

HPS-to-FPGA and Lightweight HPS-to-FPGA Bridges

Table 27-2: HPS-to-FPGA and Lightweight HPS-to-FPGA Bridges and Clocks

Interface Name	Description	Associated Clock Interface
h2f_axi_master	HPS-to-FPGA AXI master interface	h2f_axi_clock
h2f_lw_axi_master	HPS-to-FPGA lightweight AXI master interface	h2f_lw_axi_clock

The HPS-to-FPGA interface is a configurable data width AXI master (32-, 64-, or 128-bit) that allows HPS masters to issue transactions to the FPGA fabric.

The lightweight HPS-to-FPGA interface is a 32-bit AXI master that allows HPS masters to issue transactions to the FPGA fabric.

Both HPS-to-FPGA interfaces are AXI-3 compliant. The HPS-side AXI bridges manage clock crossing, buffering, and data width conversion where necessary.

Other interface standards in the FPGA fabric, such as connecting to Avalon-MM interfaces, can be supported through the use of soft logic adapters. The Platform Designer (Standard) system integration tool automatically generates adaptor logic to connect AXI to Avalon-MM interfaces.

Each AXI bridge accepts a clock input from the FPGA fabric and performs clock domain crossing internally. The exposed AXI interface operates on the same clock domain as the clock supplied by the FPGA fabric.

Related Information

- [Clocks](#) on page 27-4
 - [Features of the HPS-FPGA Bridges](#) on page 9-1
- For more information, refer to the *HPS FPGA AXI Bridges Component* chapter.

FPGA-to-HPS SDRAM Interface

The FPGA-to-HPS SDRAM interface is a direct connection between the FPGA fabric and the HPS SDRAM controller. This interface is highly configurable, allowing a mix between number of ports and port width. The interface supports both AXI-3 and Avalon-MM protocols.

Table 27-3: FPGA-to-HPS SDRAM Interfaces and Clocks

Interface Name	Description	Associated Clock Interface
f2h_sdram0_data	SDRAM AXI or Avalon-MM port 0	f2h_sdram0_clock
f2h_sdram1_data	SDRAM AXI or Avalon-MM port 1	f2h_sdram1_clock
f2h_sdram2_data	SDRAM AXI or Avalon-MM port 2	f2h_sdram2_clock
f2h_sdram3_data	SDRAM AXI or Avalon-MM port 3	f2h_sdram3_clock
f2h_sdram4_data	SDRAM AXI or Avalon-MM port 4	f2h_sdram4_clock
f2h_sdram5_data	SDRAM AXI or Avalon-MM port 5	f2h_sdram5_clock

The FPGA-to-HPS SDRAM interface is a configurable interface to the multi-port SDRAM controller.

The total data width of all interfaces is limited to a maximum of 256 bits in the read direction and 256 bits in the write direction. The interface is implemented as four 64-bit read ports and four 64-bit write ports. As a result, the minimum data width used by the interface is 64 bits, regardless of the number or type of interfaces.

You can configure this interface the following ways:

- AXI-3 or Avalon-MM protocol
- Number of interfaces
- Data width of interfaces



The FPGA-to-HPS SDRAM interface supports six command ports, allowing up to six Avalon-MM interfaces or three bidirectional AXI interfaces.

Each command port is available either to implement a read or write command port for AXI, or to form part of an Avalon-MM interface.

You can use a mix of Avalon-MM and AXI interfaces, limited by the number of command/data ports available. Some AXI features are not present in Avalon-MM interfaces.

This interface has an address width of 32 bits. To access existing Avalon-MM/AXI masters, you can use the Address Span Extender.

Note: If you connect the HPS-to-SDRAM interface with the Avalon MM pipeline bridge, you must set the bridge **Addressing Unit** setting to **Word**.

Related Information

- [Clocks](#) on page 27-4
 - [HPS Component Interfaces](#) on page 27-1
- For information about the address span extender, refer to “Using the Address Span Extender Component” in the *Instantiating the HPS Component* chapter.

Clocks

The HPS-to-FPGA clock interface supplies physical clocks to the FPGA. These clocks and resets are generated in the HPS.

Alternative Clock Inputs to HPS PLLs

This section lists alternative clock inputs to HPS PLLs.

- `f2h_periph_ref_clock`—FPGA-to-HPS peripheral PLL reference clock. You can connect this clock input to a clock in your design that is driven by the clock network on the FPGA side.
- `f2h_sdram_ref_clock`—FPGA-to-HPS SDRAM PLL reference clock. You can connect this clock to a clock in your design that is driven by the clock network on the FPGA side.

User Clocks

A user clock is a PLL output that is connected to the FPGA fabric rather than the HPS. You can connect a user clock to logic that you instantiate in the FPGA fabric.

- `h2f_user0_clock`—HPS-to-FPGA user clock, driven from main PLL
- `h2f_user1_clock`—HPS-to-FPGA user clock, driven from peripheral PLL
- `h2f_user2_clock`—HPS-to-FPGA user clock, driven from SDRAM PLL

Note: At power-up or reset, when `CSEL[3:0]=0x0` or `BSEL[2:0]` is configured to boot from FPGA, the HPS PLLs are in bypass mode. Thus, HPS user clocks exported to the FPGA fabric run at `osc1_clk` frequency. For more information refer to the *Clock Selection* section.

Related Information

[Clock Select](#) on page 29-25

The boot ROM reads the clock select values to determine what frequency has been selected for the CPU clock and any interface clock during boot.

AXI Bridge FPGA Interface Clocks

The AXI interface has an asynchronous clock crossing in the FPGA-to-HPS bridge. The FPGA-to-HPS and HPS-to-FPGA interfaces are synchronized to clocks generated in the FPGA fabric. These interfaces can be asynchronous to one another. The SDRAM controller's multiport front end (MPFE) transfers the data between the FPGA and HPS clock domains.

- `f2h_axi_clock`—AXI slave clock for FPGA-to-HPS bridge, generated in FPGA fabric
- `h2f_axi_clock`—AXI master clock for HPS-to-FPGA bridge, generated in FPGA fabric
- `h2f_lw_axi_clock`—AXI master clock for lightweight HPS-to-FPGA bridge, generated in FPGA fabric

SDRAM Clocks

You can configure the HPS component with up to six FPGA-to-HPS SDRAM clocks.

Each command channel to the SDRAM controller has an individual clock source from the FPGA fabric. The interface clock is always supplied by the FPGA fabric, with clock crossing occurring on the HPS side of the boundary.

The FPGA-to-HPS SDRAM clocks are driven by soft logic in the FPGA fabric.

- `f2h_sdram0_clock`—SDRAM clock for port 0
- `f2h_sdram1_clock`—SDRAM clock for port 1
- `f2h_sdram2_clock`—SDRAM clock for port 2
- `f2h_sdram3_clock`—SDRAM clock for port 3
- `f2h_sdram4_clock`—SDRAM clock for port 4
- `f2h_sdram5_clock`—SDRAM clock for port 5

Peripheral FPGA Clocks

The HPS peripheral clocks are exposed when the peripheral signals are routed to the FPGA.

Table 27-4: Peripheral FPGA Clocks

Clock Name	Description
<code>emac_md_clk</code>	Ethernet PHY management interface clock
<code>emac_gtx_clk</code>	Ethernet transmit clock that is used by the PHY in GMII mode
<code>emac_rx_clk_in</code>	Ethernet MAC reference clock from the PHY
<code>emac_tx_clk_in</code>	Ethernet MAC uses this clock input for TX reference

Clock Name	Description
emac_ptp_ref_clock	Ethernet timestamp precision time protocol (PTP) reference clock
qspi_sclk_out	QSPI master clock output
spim_sclk_out	SPI master serial clock output
spis_sclk_in	SPI slave serial clock input
i2c_clk	I ² C outgoing clock (part of the SCL bidirectional pin signals)
i2c_scl_in	I ² C incoming clock (part of the SCL bidirectional pin signals)

Resets

This section describes the reset interfaces to the HPS component.

Related Information

[Reset Manager](#) on page 4-1

For details about the HPS reset sequences, refer to the *Functional Description of the Reset Manager* in the *Reset Manager* chapter .

HPS-to-FPGA Reset Interfaces

The following interfaces allow the HPS to reset soft logic in the FPGA fabric:

- h2f_reset—HPS-to-FPGA warm reset
- h2f_cold_reset—HPS-to-FPGA cold reset
- h2f_warm_reset_handshake—Warm reset request and acknowledge interface between HPS and FPGA

HPS External Reset Request

The following interfaces allow soft logic in the FPGA fabric to request for different types of HPS resets:

- f2h_cold_reset_req—FPGA-to-HPS cold reset request
- f2h_warm_reset_req—FPGA-to-HPS warm reset request
- f2h_dbg_reset_req—FPGA-to-HPS debug reset request

Peripheral Reset Interfaces

The following are Ethernet reset interfaces, that can be used when the Ethernet is routed to the FPGA:

- `emac_tx_reset`— Ethernet transmit clock reset output used to reset external PHY TX clock domain logic
- `emac_rx_reset`— Ethernet receive clock reset output used to reset external PHY RX clock domain logic

Debug and Trace Interfaces

Trace Port Interface Unit

The TPIU is a bridge between on-chip trace sources and a trace port.

- `h2f_tpiu`
- `h2f_tpiu_clock_in`
- `h2f_tpiu_clock`

FPGA System Trace Macrocell Events Interface

The system trace macrocell (STM) hardware events allow logic in the FPGA to insert messages into the trace stream.

- `f2h_stm_hw_events`

FPGA Cross Trigger Interface

The cross trigger interface (CTI) allows trigger sources and sinks to interface with the embedded cross trigger (ECT).

- `h2f_cti`
- `h2f_cti_clock`

Debug APB Interface

The debug Advanced Peripheral Bus (APB) interface allows debug components in the FPGA fabric to debug components on the CoreSight debug APB.

- `h2f_debug_apb`
- `h2f_debug_apb_sideband`
- `h2f_debug_apb_reset`
- `h2f_debug_apb_clock`

Peripheral Signal Interfaces

DMA Controller Interface

The DMA controller interface allows soft IP in the FPGA fabric to communicate with the DMA controller in the HPS. You can configure up to eight separate interface channels.

- f2h_dma_req0—FPGA DMA controller peripheral request interface 0
- f2h_dma_req1—FPGA DMA controller peripheral request interface 1
- f2h_dma_req2—FPGA DMA controller peripheral request interface 2
- f2h_dma_req3—FPGA DMA controller peripheral request interface 3
- f2h_dma_req4—FPGA DMA controller peripheral request interface 4
- f2h_dma_req5—FPGA DMA controller peripheral request interface 5
- f2h_dma_req6—FPGA DMA controller peripheral request interface 6
- f2h_dma_req7—FPGA DMA controller peripheral request interface 7

Each of the DMA peripheral request interface contains the following three signals:

- f2h_dma_req—This signal is used to request burst transfer using the DMA
- f2h_dma_single—This signal is used to request single word transfer using the DMA
- f2h_dma_ack—This signal indicates the DMA acknowledgment upon requests from the FPGA

Related Information

[DMA Controller](#) on page 16-1

For details about the DMA Controller, refer to *DMA controller* chapter.

Other Interfaces

Related Information

[Cortex-A9 MPCore](#)

For more information, refer to *Cortex-A9 MPU Subsystem* .

MPU Standby and Event Interfaces

MPU standby signals are notification signals to the FPGA fabric that the MPU is in standby. Event signals are used to wake up the Cortex-A9 processors from a wait for event (WFE) state. The following shows the signals in the interface:

- h2f_mpu_events—MPU standby and event interface, including the following signals.
- h2f_mpu_eventi—Sends an event from logic in the FPGA fabric to the MPU. This FPGA-to-HPS signal is used to wake up a processor that is in a Wait For Event state. Asserting this signal has the same effect as executing the SEV instruction in the Cortex-A9. This signal must be de-asserted until the FPGA fabric is powered-up and configured.
- h2f_mpu_evento—Sends an event from the MPU to logic in the FPGA fabric. This HPS-to-FPGA signal is asserted when an SEV instruction is executed by one of the Cortex-A9processors.
- h2f_mpu_standbywfe[1:0]—Indicates which Cortex-A9 processor is in the WFE state
- h2f_mpu_standbywfi[1:0]—Indicates which Cortex-A9 processor is in the wait for interrupt (WFI) state

The MPU provides signals to indicate when it is in a standby state. These signals are available to custom hardware designs in the FPGA fabric.

Related Information

[Cortex-A9 MPCore](#)

For more information, refer to *Cortex-A9 MPU Subsystem* .

General Purpose Signals

h2f_mpu_gp—General purpose interface

FPGA-to-HPS Interrupts

You can configure the HPS component to provide 64 general purpose FPGA-to-HPS interrupts, allowing soft IP in the FPGA fabric to trigger interrupts to the MPU's generic interrupt controller (GIC). The interrupts are implemented through the following 32-bit interfaces:

- f2h_irq0—FPGA-to-HPS interrupts 0 through 31
- f2h_irq1—FPGA-to-HPS interrupts 32 through 63

The FPGA-to-HPS interrupts are asynchronous on the FPGA interface. Inside the HPS, the interrupts are synchronized to the MPU's internal peripheral clock (`periphclk`).

Related Information

[HPS Component Interfaces](#) on page 27-1

There are various HPS peripherals to FPGA interrupt interfaces that you can configure. To learn the complete list of interfaces, refer to the *Instantiating the HPS Component* chapter.

Boot from FPGA Interface

You can enable the boot from FPGA interface to indicate the availability of preloader software in the FPGA memory. This interface is used to indicate the availability of a fallback preloader software in FPGA memory as well. The fallback preloader is used when there is no valid preloader image found in HPS flash memories.

Input-only General Purpose Interface

You can enable a general purpose interface that is connected to the general purpose I/O (GPIO) peripheral of the HPS. This is an input-only interface with 14-bit wide width. The interface share the same I/O pins with the HPS DDR SDRAM controller.

Simulating the HPS Component 28

2021.05.07

av_5v4



Subscribe



Send Feedback

This section describes the simulation support for the hard processor system (HPS) component. The HPS simulation models provide bus functional models of the HPS and FPGA fabric and a simulation model of the interface to SDRAM memory.

The HPS simulation support does not include modules implemented in the HPS, such as the Arm Cortex-A9 MPCore processor.

The simulation support files are specified when the HPS component is instantiated in the Platform Designer (Standard) system integration tool. When you enable a particular HPS-FPGA interface, Platform Designer (Standard) provides the corresponding model during the generation process.

The HPS simulation support enables you to develop and verify your own FPGA user logic or intellectual property (IP) that interfaces to the HPS component.

The simulation model supports the following interfaces:

- Clock and reset interfaces
- FPGA-to-HPS Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) slave interface
- HPS-to-FPGA AXI master interface
- Lightweight HPS-to-FPGA AXI master interface
- FPGA-to-HPS SDRAM interface
- Microprocessor unit (MPU) general-purpose I/O (GPIO) interface
- MPU standby and event interface
- Interrupts interface
- Direct memory access (DMA) controller peripheral request interface
- Debug Advanced Peripheral Bus (APB) interface
- System Trace Macrocell (STM) hardware event
- FPGA cross trigger interface (CTI)
- FPGA trace port interface unit (TPIU)
- Boot from FPGA interface
- Input-only general purpose interface

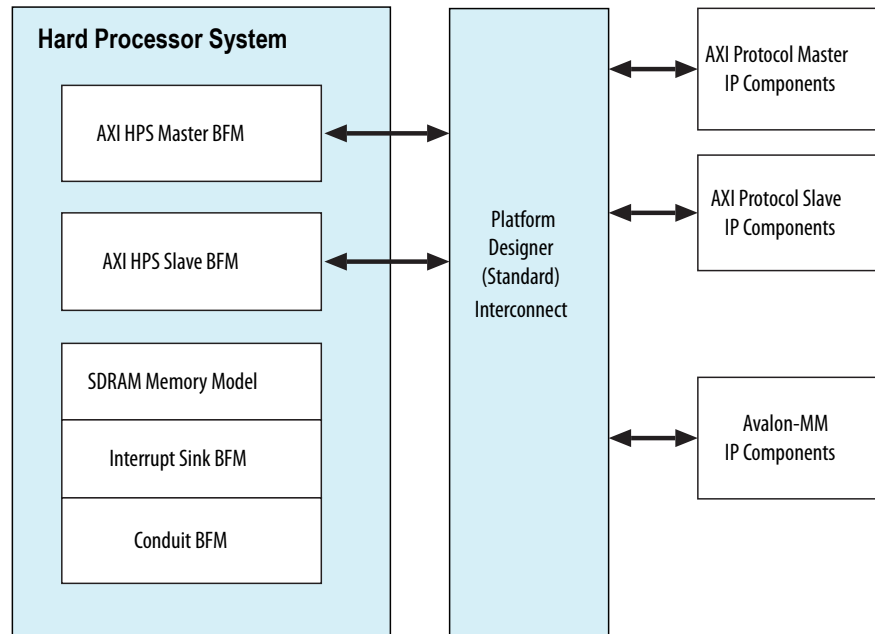
Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

ALTERA
now part of Intel

Figure 28-1: HPS BFM Block Diagram



The HPS BFM uses standard function calls from the BFM application programming interface (API), as detailed in the remainder of this chapter.

HPS simulation supports only Verilog HDL or SystemVerilog simulation environments. Users with VHDL Custom IP can run BFM simulations so long as a mixed-language simulation license is available on their chosen simulator.

Related Information

- [Simulation Flows](#) on page 28-2
- [Instantiating the HPS Component](#) on page 26-1
- [Avalon Verification IP Suite User Guide](#)
- [Mentor Graphics* Verification IP Altera Edition AMBA AXI3 and AXI4 User Guide](#)

Simulation Flows

Intel provides a functional register transfer level (RTL) simulation and a post-fitter gate-level simulation flow. Both simulation flows involve the following major steps, which is defined in the following sections:

1. Setting up the HPS component for simulation.
2. Generating the HPS simulation model in Platform Designer (Standard).
3. Running the simulation.

Related Information

[Simulating Intel FPGA Designs](#)

For general information about simulation, refer to *Simulating Intel FPGA Designs* in the *Intel Quartus Prime Standard Edition User Guide: Third-party Simulation*.

Setting Up the HPS Component for Simulation

The following steps outline how to set up the HPS component for simulation.

1. Add the HPS component from the Platform Designer (Standard) Component Library.
2. Configure the component based on your application needs by selecting or deselecting the HPS-FPGA interfaces.
3. Connect the appropriate HPS interfaces to other components in the system. For example, connect the FPGA-to-HPS AXI slave interface to an AXI or Avalon-MM master interface in another component in the system.

When you create your component, make sure the conduit interfaces have the correct role names and widths. Also make sure the conduit interfaces are opposite in direction to what is shown in the HPS Conduit Interfaces table.

Related Information

[Instantiating the HPS Component](#) on page 26-1

HPS Conduit Interfaces Connecting to the FPGA

The following tables define the HPS Conduit interfaces that connect to the FPGA.

Table 28-1: h2f_warm_reset_handshake

Role Name	Direction	Width
h2f_pending_rst_req_n	Output	1
f2h_pending_rst_ack_n	Input	1

Table 28-2: h2f_gp

Role Name	Direction	Width
h2f_gp_in	Input	32
h2f_gp_out	Output	32

Table 28-3: h2f_mpu_events

Role Name	Direction	Width
h2f_mpu_eventi	Input	1
h2f_mpu_evento	Output	1
h2f_mpu_standbywfe	Output	2
h2f_mpu_standbywfi	Output	2

Table 28-4: f2h_dma0 to f2h_dma7

Role Name	Direction	Width
f2h_dma_req<0-7>_req	Input	1
f2h_dma_req<0-7>_single	Input	1
f2h_dma_req<0-7>_ack	Output	1

Table 28-5: h2f_debug_apb_sideband

Role Name	Direction	Width
h2f_dbg_apb_PCLKEN	Input	1
h2f_dbg_apb_DBG_APB_DISABLE	Input	1

Table 28-6: f2h_stm_hw_events

Role Name	Direction	Width
f2h_stm_hwevents	Input	28

Table 28-7: h2f_cti

Role Name	Direction	Width
h2f_cti_trig_in	Input	8
h2f_cti_trig_in_ack	Output	8
h2f_cti_trig_out	Output	8
h2f_cti_trig_out_ack	Input	8
h2f_cti_fpga_clk_en	Input	1

Table 28-8: h2f_tpiu

Role Name	Direction	Width
h2f_tpiu_clk_ctrl	Input	1
h2f_tpiu_data	Output	32

Table 28-9: f2h_boot_from_fpga

Role Name	Direction	Width
f2h_boot_from_fpga_ready	Input	1
f2h_boot_from_fpga_on_failure	Input	1

Setting Up the HPS Component for Simulation

The following steps outline how to set up the HPS component for simulation.

1. Add the HPS component from the Platform Designer (Standard) Component Library.
2. Configure the component based on your application needs by selecting or deselecting the HPS-FPGA interfaces.
3. Connect the appropriate HPS interfaces to other components in the system. For example, connect the FPGA-to-HPS AXI slave interface to an AXI or Avalon-MM master interface in another component in the system.

When you create your component, make sure the conduit interfaces have the correct role names and widths. Also make sure the conduit interfaces are opposite in direction to what is shown in the HPS Conduit Interfaces table.

Related Information

[Instantiating the HPS Component](#) on page 26-1

Generating the HPS Simulation Model in Platform Designer (Standard)

The following steps outline how to generate the simulation model:

1. In Platform Designer (Standard), click **Generate HDL** under the Generate menu.
2. Choose between RTL and post-fit simulation

For RTL simulation, perform the following steps:

- a. Set **Create simulation model** to Verilog.
- b. Click **Generate**.⁽⁶²⁾

For post-fit simulation, perform the following steps:

- a. Turn on the **Create HDL design files for synthesis** option.
- b. Turn on the **Create block symbol file (.bsf)** option.⁽⁶³⁾⁽⁶⁴⁾

3. Click **Generate**.

Related Information

- [Instantiating the HPS Component](#) on page 26-1
- [Creating a System with Platform Designer \(Standard\)](#)

For more information about Platform Designer (Standard) simulation, refer to “Simulating a Platform Designer (Standard) System” in the *Intel Quartus Prime Standard Edition User Guide: Platform Designer (Standard)*.

Running the Simulations

The steps to run a simulation depend on whether you are running an RTL simulation or a post-fit simulation.

⁽⁶²⁾ VHDL is supported for HPS simulation and it requires a mix language simulator. However, the BFM's always need to be in verilog. Custom components can be in VHDL.

⁽⁶³⁾ A **.bsf** file is only needed for schematic entry.

⁽⁶⁴⁾ This is not a requirement for simulation or implementation unless a schematic is used.

Running HPS RTL Simulation

Platform Designer (Standard) generates scripts for several simulators that you can use to complete the simulation process, as listed in the following table.

Table 28-10: Platform Designer (Standard)-Generated Scripts for Supported Simulators

Simulator	Script Name	Directory
Mentor Graphics* ModelSim	msim_ setup.tcl	<i><project directory>/<design name>/simulation/mentor</i>
Cadence NC-Sim	ncsim_ setup.sh	<i><project directory>/<design name>/simulation/cadence</i>
Synopsys VCS	vcs_setup.sh	<i><project directory>/<design name>/simulation/synopsys/ vcs</i>
Synopsys VCS-MX	vcsmx_ setup.sh	<i><project directory>/<design name>/simulation/synopsys/ vcsmx</i>
Aldec* RivieraPro™	rivierapro_ setup.tcl	<i><project directory>/<design name>/simulation/aldec</i>

Related Information

- [Avalon Verification IP Suite User Guide](#)
- [Mentor Graphics Verification IP Altera Edition AMBA AXI3 and AXI4 User Guide](#)

Running HPS Post-Fit Simulation

To run HPS post-fit simulation after successful Platform Designer (Standard) generation, perform the following steps:

1. Add the generated synthesis file set to your Intel Quartus Prime project by performing the following steps:
 - a. In the Quartus Prime software, click **Settings** in the Assignments menu.
 - b. In the **Settings <your system name>** dialog box, on the **Files** tab, browse to *<your project directory>/<your system name>/synthesis/* and select *<your system name>.qip*.
 - c. Click **Open**. The **Select File** dialog box closes.
 - d. Click **OK**. The **Settings** dialog box closes.
2. Optionally instantiate your HPS system as the top-level entity in your Intel Quartus Prime project.
3. Compile the design by clicking **Start Compilation** in the Processing menu.
4. Change the EDA Netlist Writer settings, if necessary, by performing the following steps:
 - a. Click **Settings** in the Assignment menu.
 - b. On the **Simulation** tab, under the **EDA Tool Settings** tab, you can specify the following EDA Netlist Writer settings:

- **Tool name**—The name of the simulation tool
 - **Format for output netlist**
 - **Output directory**
 - c. Click **OK**.
5. To create the post-fitter simulation model with Quartus Prime EDA Netlist Writer, perform the following steps:
- a. Click **Start** in the Processing menu.
 - b. Click **Start EDA Netlist Writer**.

Related Information

[Welcome to the Intel Quartus Prime Pro Edition Software Help](#)

Post-Fit Simulation Files

Post-fit simulation is the simulation of the netlist generated from the original RTL design after it has been mapped, synthesized, and fit. The netlist represents the actual hardware and its connections as they appear in the FPGA. Quartus Prime generates the netlist and can generate a Standard Delay Format (.sdf) file with the timing information for all connections. The simulation can be functional only (without the timing information) where all wires and gates take zero time, or it can be a timing simulation where the time for all transitions is based on the SDF information.

Post-fit simulation can serve a number of different purposes. It can be used to perform a dynamic verification of the timing of the design, or it can be used to verify the functional correctness of either the design, the compilation flow (in particular, the fitter), or both.

This table uses the following symbols:

- *<ACDS install>* = Intel SoC FPGA Embedded Development Suite installation path
- *<Avalon Verification IP>* = *<ACDS install>/ip/altera/sopc_builder_ip/verification*
- *<AXI Verification IP>* = *<ACDS install>/ip/altera/mentor_vip_ae*
- *<HPS Post-fit Sim>* = *<ACDS install>/ip/altera/hps/postfitter_simulation*
- *<Device Sim Lib>* = *<ACDS install>/quartus/eda/sim_lib*

Table 28-11: Post-Fit Simulation Files

Library	Directory	File
Altera Verification IP Library	<i><Avalon Verification IP>/lib/</i>	verbosity_pkg.sv avalon_mm_pkg.sv avalon_utilities_pkg.sv
Avalon Clock Source BFM	<i><Avalon Verification IP>/altera_avalon_clock_source/</i>	altera_avalon_clock_source.sv
Avalon Reset Source BFM	<i><Avalon Verification IP>/altera_avalon_reset_source/</i>	altera_avalon_reset_source.sv
Avalon MM Slave BFM	<i><Avalon Verification IP>/altera_avalon_mm_slave_bfm/</i>	altera_avalon_mm_slave_bfm.sv

Library	Directory	File
Avalon Interrupt Sink BFM	<Avalon Verification IP>/altera_avalon_interrupt_sink/	altera_avalon_interrupt_sink.sv
Mentor AXI Verification IP Library	<AXI Verification IP>/common/	questa_mvc_svapi.svh
Mentor AXI3 BFM	<AXI Verification IP>/axi3/axi3/bfm/	mgc_common_axi.sv mgc_axi_master.sv mgc_axi_slave.sv
HPS Post-Fit Simulation Library	<HPS Post-fit Sim>/	All the files in the directory
Device Simulation Library ⁽⁶⁵⁾	<Device Sim Lib>/	altera_primitives.v 220model.v sgate.v altera_mf.v altera_lnsim.sv cyclonev_atoms.v arriav_atoms.v mentor/cyclonev_atoms_ncrypt.v mentor/arriav_atoms_ncrypt.v
EDA Netlist Writer Generated Post-Fit Simulation Model	<User project directory>/	*.vo *.vho (Mixed-language simulator is needed for Verilog HDL and VHDL mixed design)
User testbench files	<User project directory>/	*.v *.sv *.vhd (Mixed-language simulator is needed for Verilog HDL and VHDL mixed design)

⁽⁶⁵⁾ The device simulation library is not needed with .

BFM API Hierarchy Format

For post-fit simulation, you must call the BFM API in your test program with a specific hierarchy. The hierarchy format is:

```
<DUT>.\<HPS>|fpga_interfaces|<interface><space>.<BFM>.<API function>
```

Where:

- **<DUT>** is the instance name of the design under test that you instantiated in your test bench. The design under test is the HPS component.
- **<HPS>** is the HPS component instance name that you use in your Platform Designer (Standard) system.
- **<interface>** is the instance name of a specific FPGA-to-HPS or HPS-to-FPGA interface. This name can be found in the **fpga_interfaces.sv** file located in **<project directory>/<design name>/synthesis/submodules**.
- **<space>**—You must insert one space character after the interface instance name.
- **<BFM>** is the BFM instance name. To identify the BFM instance name, in **<ACDS install>/ip/altera/hps/postfitter_simulation**, find the SystemVerilog file corresponding to the interface type that you are using. This SystemVerilog file contains the BFM instance name.

For example, a path for the Lightweight HPS-to-FPGA master interface hierarchy could be formed as follows:

```
top.dut.\my_hps_component|fpga_interface|hps2fpga_light_weight .h2f_lw_axi_master
```

Notice the space after **hps2fpga_light_weight**. Omitting this space would cause simulation failure because the instance name **hps2fpga_light_weight**, including the space, is the name used in the post-fit simulation model generated by the Intel Quartus Prime software.

Clock and Reset Interfaces

Clock Interface

Platform Designer (Standard) generates the BFM clock for each clock input interface from the FPGA component. For the FPGA-to-HPS PLL reference clocks, specify the BFM reference clock frequency in the **Reference clock frequency** field in the **HPS Clocks** page when instantiating the HPS component in Platform Designer (Standard).

Table 28-12: HPS Clock Input Interface Simulation Model

The Intel clock source BFM application programming interface (API) applies to all the BFM instances listed in this table. Your Verilog interfaces use the same API.

Interface Name	BFM Instance Name
f2h_periph_ref_clock	f2h_periph_ref_clock_inst
f2h_sdram_ref_clock	f2h_sdram_ref_clock_inst

Platform Designer (Standard) generates the clock source BFM for each clock output interface from the HPS component. For HPS-to-FPGA user clocks, specify the BFM clock rate in the **User clock frequency** field in the **HPS Clocks** page when instantiating the HPS component in Platform Designer (Standard).

The HPS-to-FPGA TPIU generates a clock output to the FPGA, named `h2f_tpiu_clock`. In simulation, the clock source BFM also represents this clock output's behavior. Also, the HPS-to-FPGA debug APB interface generates a clock output to the FPGA, named `h2f_debug_apb_clock`.

Table 28-13: HPS Clock Output Interface Simulation Model

The Intel clock source BFM application programming interface (API) applies to all the BFM's listed in this table. Your Verilog interfaces use the same API.

Interface Name	BFM Instance Name
<code>h2f_user0_clock</code>	<code>h2f_user0_clock_inst</code>
<code>h2f_user1_clock</code>	<code>h2f_user1_clock_inst</code>
<code>h2f_user2_clock</code>	<code>h2f_user2_clock_inst</code>
<code>h2f_tpiu_clock</code>	<code>h2f_tpiu_clock_inst</code>

Related Information

[Memory-Mapped Interfaces](#) on page 27-1

Reset Interface

The HPS reset request and handshake interfaces are connected to conduit BFM's for simulation.

Table 28-14: HPS Reset Input Interface Simulation Model

You can monitor the reset request interface state changes or set the interface by using the API listed.

Interface Name	BFM Instance Name	API Function Names
<code>f2h_cold_reset_req</code>	<code>f2h_cold_reset_req_inst</code>	<code>get_f2h_cold_rst_req_n()</code>
<code>f2h_debug_reset_req</code>	<code>f2h_debug_reset_req_inst</code>	<code>get_f2h_dbg_rst_req_n()</code>
<code>f2h_warm_reset_req</code>	<code>f2h_warm_reset_req_inst</code>	<code>get_f2h_warm_rst_req_n()</code>
<code>h2f_warm_reset_handshake</code>	<code>h2f_warm_reset_handshake_inst</code>	<code>set_h2f_pending_rst_req_n()</code> <code>get_f2h_pending_rst_ack_n()</code>

Table 28-15: HPS Reset Output Interface Simulation Model

The reset source BFM application programming interface applies to all the BFM's listed.

Interface Name	BFM Instance Name
<code>h2f_reset</code>	<code>h2f_reset_inst</code>
<code>h2f_cold_reset</code>	<code>h2f_cold_reset_inst</code>
<code>h2f_debug_apb_reset</code>	<code>h2f_debug_apb_reset_inst</code>

Table 28-16: Configuration of Reset Source BFM for HPS Reset Output Interface

The HPS reset output interface is connected to a reset source BFM. Platform Designer (Standard) configures the BFM as shown in the following table. The parameter value of the instantiated BFM is configured for HPS simulation.

Parameter	BFM Value	Meaning
Assert reset high	Off	This parameter is off, specifying an active-low reset signal from the BFM.
Cycles of initial reset	0	This parameter is 0, specifying that the BFM does not assert the reset signal automatically.

Related Information

- [Memory-Mapped Interfaces](#) on page 27-1
- [Avalon Verification IP Suite User Guide](#)

FPGA-to-HPS AXI Slave Interface

The FPGA-to-HPS AXI slave interface, `f2h_axi_slave`, is connected to a Mentor Graphics AXI slave BFM for simulation with an instance name of `f2h_axi_slave_inst`. Platform Designer (Standard) configures the BFM as shown in the following table. The BFM clock input is connected to `f2h_axi_clock`.

Table 28-17: Configuration of FPGA-to-HPS AXI Slave BFM

Parameter	Value
AXI Address Width	32
AXI Read Data Width	32, 64, or 128
AXI Write Data Width	32, 64, or 128
AXI ID Width	8

You control and monitor the AXI slave BFM by using the BFM API.

Related Information

- [Memory-Mapped Interfaces](#) on page 27-1
- [Mentor Graphics Verification IP Altera Edition AMBA AXI3 and AXI4 User Guide](#)
The Mentor Graphics Verification IP User guide provides details of the API and connection guidelines for the AXI3 and AXI4 BFM.

HPS-to-FPGA AXI Master Interface

The HPS-to-FPGA AXI master interface, `h2f_axi_master`, is connected to a Mentor Graphics AXI master BFM for simulation with an instance name of `h2f_axi_master_inst`. In Platform Designer (Standard), you can configure the HPS-to-FPGA interface with the following address, data, and ID widths. The BFM clock input is connected to `h2f_axi_clock` clock.

Table 28-18: Configuration of HPS-to-FPGA AXI Master BFM

Parameter	Value
AXI Address Width	30
AXI Read and Write Data Width	32, 64, or 128
AXI ID Width	12

You control and monitor the AXI master BFM by using the BFM API.

Related Information

- [Memory-Mapped Interfaces](#) on page 27-1
- [Mentor Graphics Verification IP Altera Edition AMBA AXI3 and AXI4 User Guide](#)
The Mentor Graphics Verification IP User guide provides details of the API and connection guidelines for the AXI3 and AXI4 BFM.

Lightweight HPS-to-FPGA AXI Master Interface

The lightweight HPS-to-FPGA AXI master interface, `h2f_lw_axi_master`, is connected to a Mentor Graphics AXI master BFM for simulation with an instance name of `h2f_lw_axi_master_inst`. Platform Designer (Standard) configures the BFM as shown in the following table. The BFM clock input is connected to `h2f_lw_axi_clock` clock.

Table 28-19: Configuration of Lightweight HPS-to-FPGA AXI Master BFM

Parameter	Value
AXI Address Width	21
AXI Read and Write Data Width	32
AXI ID Width	12

You control and monitor the AXI master BFM by using the BFM API.

Related Information

- [Memory-Mapped Interfaces](#) on page 27-1

- **Mentor Graphics Verification IP Altera Edition AMBA AXI3 and AXI4 User Guide**
The Mentor Graphics Verification IP User guide provides details of the API and connection guidelines for the AXI3 and AXI4 BFM.

FPGA-to-HPS SDRAM Interface

The HPS component contains a memory interface simulation model to which all of the FPGA-to-HPS SDRAM interfaces are connected. The model is based on the HPS implementation and provides cycle-level accuracy, reflecting the true bandwidth and latency of the interface. However, the model does not have the detailed configuration provided by the HPS software, and hence does not reflect any inter-port scheduling that might occur under contention on the real hardware when different priorities or weights are used.

Related Information

External Memory Interface Handbook Volume 3: Reference Material

For more information, refer to the *Functional Description—Hard Memory Interface* section

HPS Memory Interface Simulation

The HPS component provides a complete simulation model of the HPS memory interface controller and PHY, with cycle-level accuracy, comparable to the simulation models for the FPGA memory interface.

There are three simulation modes:

- Skip-cal - In the simulation, the memory comes up already calibrated.
- Quick-cal - The calibration process is sped up so that you do not have to wait so long into the simulation to start reading or writing memory.
- Full-cal - The entire calibration sequence is simulated. This means that you can be waiting a while before the memory controller is online.

The simulation model supports only the skip-cal simulation mode. Quick-cal and full-cal are not supported. Although an example design is not provided, you can create a test design by adding the traffic generator component to your design using Platform Designer (Standard). Also, the HPS simulation model does not use external memory pins to connect to the DDR memory model; instead, the memory model is incorporated directly into the HPS SDRAM interface simulation modules.

One of the various ways you can simulate the FPGA-to-SDRAM interfaces is by:

- Bring the interfaces out of reset; otherwise, transactions cannot occur.
- Connect the H2F reset to the F2H port resets.
- Add a stage to your testbench to assert and deassert the H2F reset in the HPS.

The appropriate Verilog code is shown below:

```
Initial
begin
    // Assert reset
    <base name>.hps.fpga_interfaces.h2f_reset_inst.reset_assert();
    // Delay
    #1
    // Deassert reset
    <base name>.hps.fpga_interfaces.h2f_reset_inst.reset_deassert();
end
```



HPS-to-FPGA MPU Event Interface

The HPS-to-FPGA MPU event interface is connected to a conduit BFM for simulation. The following table lists the name of each interface, along with API function names for each type of simulation. You can monitor the interface state changes or set the interface by using the API listed.

Table 28-20: HPS-to-FPGA MPU Event Interface Simulation Model

The usage of conduit `get_*()` and `set_*()` API functions is the same as with the general Avalon conduit BFM.

Interface Name	BFM Instance Name	RTL Simulation API Function Names	Post-Fit Simulation API Function Names
h2f_mpu_events	h2f_mpu_events_inst	<code>get_h2f_mpu_eventi()</code> <code>set_h2f_mpu_evento()</code> <code>set_h2f_mpu_standbywfe()</code> <code>set_h2f_mpu_standbywfi()</code>	<code>get_eventi()</code> <code>set_evento()</code> <code>set_standbywfe()</code> <code>set_standbywfi()</code>

Related Information

[Avalon Verification IP Suite User Guide](#)

Interrupts Interface

The FPGA-to-HPS interrupts interface is connected to an Avalon interrupt sink BFM for simulation.

Table 28-21: FPGA-to-HPS Interrupts Interface Simulation Model

Interface Name	BFM Instance Name
f2h_irq0	f2h_irq0_inst
f2h_irq1	f2h_irq1_inst

The HPS-to-FPGA peripheral interfaces are connected to conduit BFMs for simulation. When you enable the peripheral interrupt, the corresponding peripheral signal to the FPGA is exposed.

Table 28-22: HPS-to-FPGA Peripherals Interrupt Interface Simulation Model

Interface Name	BFM Instance Name
h2f_can0_interrupt	h2f_can0_interrupt_inst
h2f_can1_interrupt	h2f_can1_interrupt_inst
h2f_clkmgr_interrupt	h2f_clkmgr_interrupt_inst

Interface Name	BFM Instance Name
h2f_mpuwakeup_interrupt	h2f_mpuwakeup_interrupt_inst
h2f_cti_interrupt0	h2f_cti_interrupt0_inst
h2f_cti_interrupt1	h2f_cti_interrupt1_inst
h2f_dma_interrupt0	h2f_dma_interrupt0_inst
h2f_dma_interrupt1	h2f_dma_interrupt1_inst
h2f_dma_interrupt2	h2f_dma_interrupt2_inst
h2f_dma_interrupt3	h2f_dma_interrupt3_inst
h2f_dma_interrupt4	h2f_dma_interrupt4_inst
h2f_dma_interrupt5	h2f_dma_interrupt5_inst
h2f_dma_interrupt6	h2f_dma_interrupt6_inst
h2f_dma_interrupt7	h2f_dma_interrupt7_inst
h2f_dma_abort_interrupt	h2f_dma_abort_interrupt_inst
h2f_emac0_interrupt	h2f_emac0_interrupt_inst
h2f_emac1_interrupt	h2f_emac1_interrupt_inst
h2f_fpga_man_interrupt	h2f_fpga_man_interrupt_inst
h2f_gpio0_interrupt	h2f_gpio0_interrupt_inst
h2f_gpio1_interrupt	h2f_gpio1_interrupt_inst
h2f_gpio2_interrupt	h2f_gpio2_interrupt_inst
h2f_i2c_emac0_interrupt	h2f_i2c_emac0_interrupt_inst
h2f_i2c_emac1_interrupt	h2f_i2c_emac1_interrupt_inst
h2f_i2c0_interrupt	h2f_i2c0_interrupt_inst
h2f_i2c1_interrupt	h2f_i2c1_interrupt_inst
h2f_l4sp0_interrupt	h2f_l4sp0_interrupt_inst
h2f_l4sp1_interrupt	h2f_l4sp1_interrupt_inst
h2f_nand_interrupt	h2f_nand_interrupt_inst
h2f_osc0_interrupt	h2f_osc0_interrupt_inst
h2f_osc1_interrupt	h2f_osc1_interrupt_inst
h2f_qspi_interrupt	h2f_qspi_interrupt_inst
h2f_sdmmc_interrupt	h2f_sdmmc_interrupt_inst
h2f_spi0_interrupt	h2f_spi0_interrupt_inst
h2f_spi1_interrupt	h2f_spi1_interrupt_inst
h2f_spi2_interrupt	h2f_spi2_interrupt_inst
h2f_spi3_interrupt	h2f_spi3_interrupt_inst
h2f_usb0_interrupt	h2f_usb0_interrupt_inst

Interface Name	BFM Instance Name
h2f_usb1_interrupt	h2f_usb1_interrupt_inst
h2f_wdog0_interrupt	h2f_wdog0_interrupt_inst
h2f_wdog1_interrupt	h2f_wdog1_interrupt_inst
h2f_uart0_interrupt	h2f_uart0_interrupt_inst
h2f_uart1_interrupt	h2f_uart1_interrupt_inst

HPS-to-FPGA Debug APB Interface

The HPS-to-FPGA debug APB interface is connected to an Intel conduit BFM for simulation. The following table lists the name of each interface, along with API function names for each type of simulation. You can monitor the interface state changes or set the interface by using the API functions listed.

Table 28-23: HPS-to-FPGA Debug APB Interface Simulation Model

Interface Name	BFM Name	RTL Simulation API Function Names	Post-Fit Simulation API Function Names
h2f_debug_apb	h2f_debug_apb	set_h2f_dbg_apb_PADDR() set_h2f_dbg_apb_PADDR_31() set_h2f_dbg_apb_PENABLE() get_h2f_dbg_apb_PRDATA() get_h2f_dbg_apb_PREADY() set_h2f_dbg_apb_PSEL() get_h2f_dbg_apb_PSLVERR() set_h2f_dbg_apb_PWDATA() set_h2f_dbg_apb_PWRITE()	set_PADDR() set_PADDR_31() set_PENABLE() get_PRDATA() get_PREADY() set_PSEL() get_PSLVERR() set_PWDATA() set_PWRITE()
h2f_debug_apb_sideband	h2f_debug_apb_sideband	get_h2f_dbg_apb_PCLKEN() get_h2f_dbg_apb_DBG_APB_DISABLE()	get_PCLKEN() get_DBG_APB_DISABLE()

FPGA-to-HPS System Trace Macrocell Hardware Event Interface

The FPGA-to-HPS STM hardware event interface is connected to a conduit BFM for simulation. The following table lists the name of each interface, along with the API function name for each type of simulation. You can monitor the interface state changes or set the interface by using the API functions listed.

Table 28-24: FPGA-to-HPS STM Hardware Event Interface Simulation Model

Interface Name	BFM Name	RTL Simulation API Function Name	Post-Fit Simulation API Function Name
f2h_stm_hw_events	f2h_stm_hw_events_inst	get_f2h_stm_hwevents()	get_stm_events()

HPS-to-FPGA Cross-Trigger Interface

The HPS-to-FPGA cross-trigger interface is connected to a conduit BFM for simulation. The following table lists the name of each interface, along with API function names for each type of simulation. You can monitor the interface state changes or set the interface by using the API functions listed.

Table 28-25: HPS-to-FPGA Cross-Trigger Interface Simulation Model

Interface Name	BFM Name	RTL Simulation API Function Names	Post-Fit Simulation API Function Names
h2f_cti	h2f_cti_inst	get_h2f_cti_trig_in()	get_trig_in()
		set_h2f_cti_trig_in_ack()	set_trig_inack()
		set_h2f_cti_trig_out()	set_trig_out()
		get_h2f_cti_trig_out_ack()	get_trig_outack()
		get_h2f_cti_fpga_clk_en()	get_clk_en()

HPS-to-FPGA Trace Port Interface

The HPS-to-FPGA trace port interface is connected to a conduit BFM for simulation. The following table lists the name of each interface, along with API function names for each type of simulation. You can monitor the interface state changes or set the interface by using the API functions listed.

Table 28-26: HPS-to-FPGA Trace Port Interface Simulation Model

Interface Name	BFM Name	RTL Simulation API Function Names	Post-Fit Simulation API Function Names
h2f_tpiu	h2f_tpiu_inst	get_h2f_tpiu_clk_ctl()	get_traceclk_ctl()
		set_h2f_tpiu_data()	set_trace_data()

FPGA-to-HPS DMA Handshake Interface

The FPGA-to-HPS DMA handshake interface is connected to a conduit BFM for simulation. The following table lists the name for each interface, along with API function names for each type of simulation. You can monitor the interface state changes or set the interface by using the API listed.

Table 28-27: FPGA-to-HPS DMA Handshake Interface Simulation Model

The usage of conduit `get_*()` and `set_*()` API functions is the same as with the general Avalon conduit BFM.

Interface Name	BFM Instance Name	RTL Simulation API Function Names	Post-Fit Simulation API Function Names
f2h_dma0	f2h_dma0_inst	get_f2h_dma0_req() get_f2h_dma0_single() set_f2h_dma0_ack()	get_channel0_req() get_channel0_single() set_channel0_xx_ack()
f2h_dma1	f2h_dma1_inst	get_f2h_dma1_req() get_f2h_dma1_single() set_f2h_dma1_ack()	get_channel1_req() get_channel1_single() set_channel1_xx_ack()
f2h_dma2	f2h_dma2_inst	get_f2h_dma2_req() get_f2h_dma2_single() set_f2h_dma2_ack()	get_channel2_req() get_channel2_single() set_channel2_xx_ack()
f2h_dma3	f2h_dma3_inst	get_f2h_dma3_req() get_f2h_dma3_single() set_f2h_dma3_ack()	get_channel3_req() get_channel3_single() set_channel3_xx_ack()

Interface Name	BFM Instance Name	RTL Simulation API Function Names	Post-Fit Simulation API Function Names
f2h_dma4	f2h_dma4_inst	get_f2h_dma4_req() get_f2h_dma4_single() set_f2h_dma4_ack()	get_channel4_req() get_channel4_single() set_channel4_xx_ack()
f2h_dma5	f2h_dma5_inst	get_f2h_dma5_req() get_f2h_dma5_single() set_f2h_dma5_ack()	get_channel5_req() get_channel5_single() set_channel5_xx_ack()
f2h_dma6	f2h_dma6_inst	get_f2h_dma6_req() get_f2h_dma6_single() set_f2h_dma6_ack()	get_channel6_req() get_channel6_single() set_channel6_xx_ack()
f2h_dma7	f2h_dma7_inst	get_f2h_dma7_req() get_f2h_dma7_single() set_f2h_dma7_ack()	get_channel7_req() get_channel7_single() set_channel7_xx_ack()

Related Information

[Avalon Verification IP Suite User Guide](#)

Boot from FPGA Interface

The boot from FPGA interface is connected to a conduit BFM for simulation. You can monitor the interface state changes or set the interface by using the API functions in the table below.

Table 28-28: Boot from FPGA Interface Simulation Model

Interface Name	BFM Name	RTL simulation API Function Names	Post-fit Simulation API Function Names
f2h_boot_from_fpga	f2h_boot_from_fpga_inst	get_f2h_boot_from_fpga_ready() get_f2h_boot_from_fpga_on_failure()	get_boot_fpga_ready() get_boot_from_fpga_on_failure()

General Purpose Input Interface

The general purpose input (GPI) interface is connected to a conduit BFM for simulation. You can monitor the interface state changes or set the interface by using the API functions in the table below.

Table 28-29: General Purpose Input Interface Simulation Model

Interface Name	BFM Name	RTL simulation API function names
hps_io	hps_io_inst	get_hps_io_gpio_inst_HLGPI0() get_hps_io_gpio_inst_HLGPI1() get_hps_io_gpio_inst_HLGPI2() get_hps_io_gpio_inst_HLGPI3() get_hps_io_gpio_inst_HLGPI4() get_hps_io_gpio_inst_HLGPI5() get_hps_io_gpio_inst_HLGPI6() get_hps_io_gpio_inst_HLGPI7() get_hps_io_gpio_inst_HLGPI8() get_hps_io_gpio_inst_HLGPI9() get_hps_io_gpio_inst_HLGPI10() get_hps_io_gpio_inst_HLGPI11() get_hps_io_gpio_inst_HLGPI12() get_hps_io_gpio_inst_HLGPI13()

2021.05.07

av_5v4



Subscribe



Send Feedback

The Intel system-on-a-chip (SoC) device provides a variety of ways to boot the hard processor system (HPS) and configure the FPGA portion.

Boot Overview

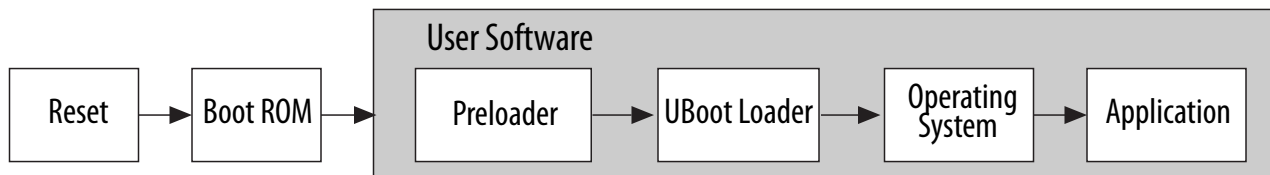
The boot flow of the HPS is a multi-stage process. Each stage is responsible for loading the next stage.

The first software stage is the boot ROM. The boot ROM code locates and executes the second software stage, called the preloader. The preloader locates, and if present, executes the next software stage. The preloader and subsequent software stages (if present) are collectively referred to as user software.

Only the boot ROM code is located in the HPS. Subsequent software is located external to the HPS and is provided by users. The boot ROM code is only aware of the preloader and not aware of any potential subsequent software stages.

The figure below illustrates the typical boot flow. However, there may be more or less software stages in the user software than shown and the roles of the software stages may vary.

Figure A-1: Typical Boot Flow



FPGA Configuration Overview

After power is applied to the FPGA and HPS portion of the SoC, configuration may begin. The FPGA may be configured through a variety of ways, such as through the HPS, JTAG, or an external host.

The control block (CB) in the FPGA portion of the device is responsible for obtaining an FPGA configuration image and configuring the FPGA. The FPGA configuration ends when the configuration image has been fully loaded and the FPGA enters user mode. The FPGA configuration image is provided by users and is typically stored in external non-volatile flash-based memory. The FPGA CB can obtain a configura-

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2015
Registered

tion image from the HPS through the FPGA manager or from any of the sources supported by the Arria V FPGA family.

Related Information

- [FPGA Manager](#) on page 5-1
For more information regarding programming the FPGA through the HPS, refer to the *FPGA Manager* chapter.
- [Configuration, Design Security, and Remote System Upgrades in Arria V Devices](#)

Bootling and Configuration Options

SoC initialization includes the bootling of the HPS and the configuration of the FPGA fabric and I/O.

Depending on the initialization option you choose, the I/O configuration is handled differently. You can choose one of the three initialization options:

- The HPS boot and FPGA configuration occur separately.
- The HPS boots first and then configures the FPGA.
- The HPS boots from the FPGA after the FPGA is configured.

Note: The HPS and FPGA fabric must be powered at the same time. You must not power-down the HPS or FPGA independently in the middle of device operation.

The following three figures illustrate the possible HPS boot and FPGA configuration schemes. The arrows in the figures denote the data flow direction.

Figure A-2: Separate FPGA Configuration and HPS Booting

In the figure below, the FPGA configuration and HPS boot can occur independently. The FPGA obtains its configuration image from a non-HPS source, while the HPS boot ROM obtains its preloader from a non-FPGA fabric source.

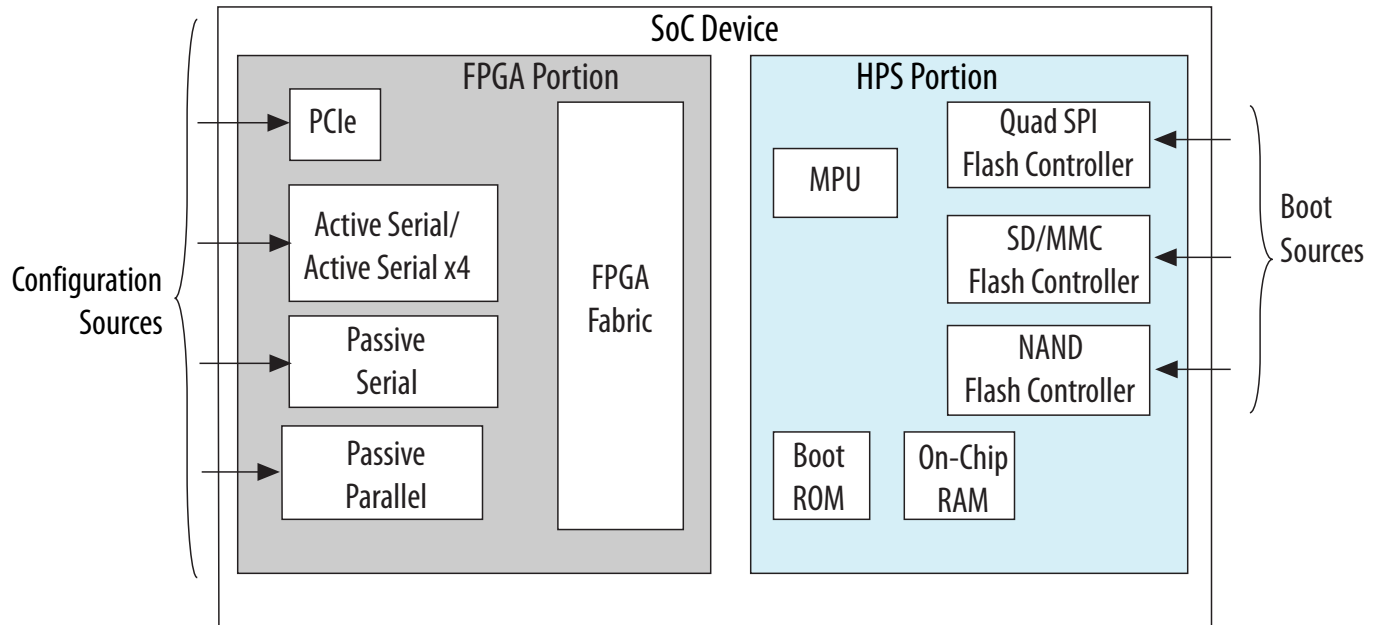


Figure A-3: HPS Boots First and then Configures the FPGA

In the figure below, the HPS boots first through one of its non-FPGA fabric boot sources. The FPGA does not need to be configured in order for the HPS to boot. However, the FPGA must be in a power-on state for the HPS to reset properly. The software on the HPS obtains the FPGA configuration image from any of its flash memory devices or communication interfaces, for example, the Ethernet media access controller (EMAC).

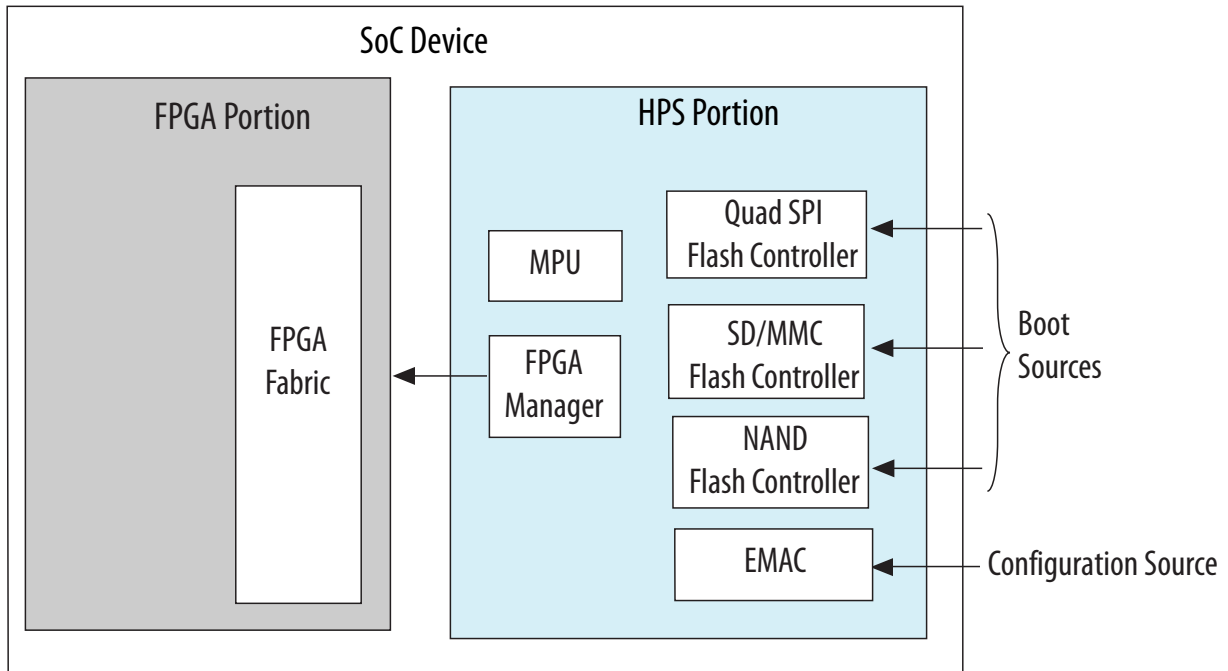
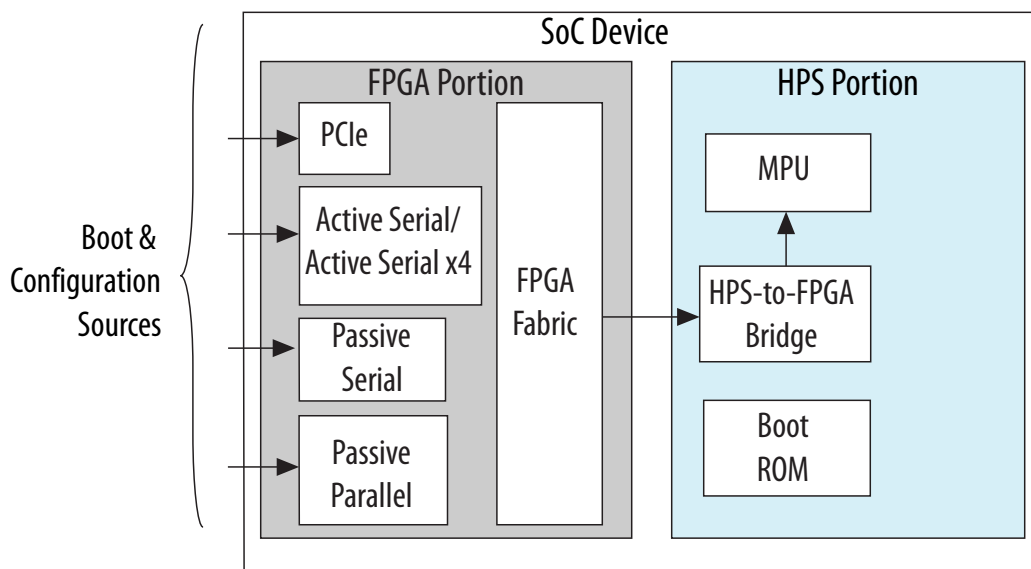


Figure A-4: HPS Boots From FPGA

In the figure below, the FPGA is configured first through one of its non-HPS configuration sources and then the HPS executes the preloader from the FPGA. In this situation, the HPS should not be released from reset until the FPGA is powered on and programmed. Once the FPGA is in user mode and the HPS has been released from reset, the boot ROM code begins executing. The HPS boot ROM code executes the preloader from the FPGA fabric over the HPS-to-FPGA bridge.



Boot Definitions

The following sections contain basic terms and definitions that are part of the boot process.

Reset

Reset precedes the boot stages and is an important part of device initialization. There are two different reset types: cold reset and warm reset.

The boot process begins when CPU0 in the MPU exits from the reset state. When CPU0 exits from reset, it starts executing code at the reset exception address where the boot ROM code is located. CPU1 remains in reset during this time and is brought out of reset by user software.

With warm reset, some software registers are preserved and the boot process may skip some steps depending on software settings. In addition, on a warm reset, the preloader has the ability to be executed from on-chip RAM.

Boot ROM

The boot ROM code is 64 KB in size and located in on-chip ROM at address range 0xFFFD0000 to 0xFFDFFFFF. The function of the boot ROM code is to determine the boot source, initialize the HPS after a reset, and jump to the preloader. In the case of indirect execution, the boot ROM code loads the

preloader image from the flash memory to on-chip RAM. The boot ROM performs the following actions to initialize the HPS:

- Enable instruction cache, branch predictor, floating point unit, NEON vector unit of CPU0
- Sets up the level 4 (L4) watchdog 0 timer
- Configures the main PLL and peripheral PLL based on the CSEL value
- Initializes the flash controller to default settings

When booting from flash memory, the boot ROM code uses the top 4 KB of the on-chip RAM as data workspace. This area is reserved for the boot ROM code after a reset until the boot ROM code passes software control to preloader. For a warm RAM boot or a cold boot from FPGA, the boot ROM code does not reserve the top 4 KB of the on-chip RAM, and the user may place user data in this area without being overwritten by boot ROM.

The boot process begins when CPU0 exits from the reset state. The boot ROM code only executes on CPU0. CPU1 is held in reset while boot ROM executes on CPU0. When a CPU0 exits from reset, it starts executing code at the reset exception address.

When CPU0 exits the boot ROM code and starts executing user software, the boot ROM access is disabled. The user software in CPU0 must map the user software exception vectors at 0x0 (which is previously mapped to boot ROM exception vectors). The user software also has the option of releasing CPU1 from reset. If CPU1 is released from reset, CPU1 executes the user software exception instead of boot ROM.

Boot Select

The boot select (BSEL) pins offer multiple methods to obtain the preloader image. On a cold reset or when a RAM boot has not been requested, the user asserts the BSEL pins on the device to indicate the boot source that is required. These pins are sampled on deassertion of cold reset and are written to the `bootinfo` register in the System Manager. The value of this register is read during boot ROM execution so that the appropriate interface pins can be initialized. If the boot source is from FPGA, the `bse1` field value reads as 0x1 in the `bootinfo` register and no HPS interface is configured for external boot.

Note: The acronyms BSEL and BOOTSEL are used interchangeably to define the boot select pins.

Table A-1: BSEL Values for Boot Source Selection

BSEL[2:0] Value	Flash Device
0x0	Reserved
0x1	FPGA (HPS-to-FPGA bridge)
0x2	1.8 V NAND flash memory
0x3	3.3 V NAND flash memory
0x4	1.8 V SD/MMC flash memory with external transceiver
0x5	3.3 V SD/MMC flash memory with internal transceiver

BSEL[2:0] Value	Flash Device
0x6	1.8 V quad SPI flash memory
0x7	3.3 V quad SPI flash memory

Note: If the BSEL value is set to 0x4 or 0x5, an external translation transceiver may be required to supply level-shifting and isolation if the SD cards interfacing to the SD/MMC controller must operate at a different voltage than the controller interface. Please refer to the *SD/MMC Controller* chapter for more information.

The HPS flash sources can store various file types, such as:

- FPGA programming files
- Preloader binary file (up to four copies)
- Boot loader binary file
- Operating system binary files
- Application file system

When BSEL = 0x1, the boot source is the FPGA and the CSEL pins are ignored. PLLs are bypassed so that OSC1 drives all the clocks.

If an HPS flash interface has been selected to load the boot image, then the boot ROM enables and configures that interface before loading the boot image into on-chip RAM, verifying it and passing software control to the preloader.

If the external RAM boot source is invalid or if the boot ROM code cannot find a valid image in flash, then the boot ROM code checks to see if there is a fallback image in the FPGA. If there is then the boot ROM executes that.

If the FPGA fabric is the boot source, the boot ROM code waits until the FPGA portion of the device is in user mode, and is ready to execute code and then passes software control to the preloader in the FPGA RAM.

Related Information

[SD/MMC Controller](#) on page 14-1

Refer to the *SD/MMC Controller* chapter for more information regarding features and functionality.

Boot Source I/O Mapping

The table below shows how the boot source signals are mapped to the I/O pins when the BSEL pins are configured.

Table A-2: Boot Source I/O Mappings

Note: "-" means that the pin is not used for that boot select interface.

Pin	Boot Interface Signals		
	NAND (BSEL[2:0] = 0x2 or 0x3)	SD/MMC (BSEL[2:0] = 0x4 or 0x5)	QSPI (BSEL[2:0] = 0x6 or 0x7)
MIXED1I00	NAND_ALE	-	-
MIXED1I01	NAND_CE	-	-

Pin	Boot Interface Signals		
	NAND (BSEL[2:0] = 0x2 or 0x3)	SD/MMC (BSEL[2:0] = 0x4 or 0x5)	QSPI (BSEL[2:0] = 0x6 or 0x7)
MIXED1I02	NAND_CLE	-	-
MIXED1I03	NAND_RE	-	-
MIXED1I04	NAND_RB	-	-
MIXED1I05	NAND_DQ0	-	-
MIXED1I06	NAND_DQ1	-	-
MIXED1I07	NAND_DQ2	-	-
MIXED1I08	NAND_DQ3	-	-
MIXED1I09	NAND_DQ4	-	-
MIXED1I010	NAND_DQ5	-	-
MIXED1I011	NAND_DQ6	-	-
MIXED1I012	NAND_DQ7	-	-
MIXED1I013	NAND_WP	-	-
MIXED1I014	NAND_WE	-	-
MIXED1I015	-	-	QSPI_I00
MIXED1I016	-	-	QSPI_I01
MIXED1I017	-	-	QSPI_I02
MIXED1I018	-	-	QSPI_I03
MIXED1I019	-	-	QSPI_SS0
MIXED1I020	-	-	QSPI_CLK
MIXED1I021	-	-	QSPI_SS1
FLASHI00	-	SDMMC_CMD	-
FLASHI01	-	-	-
FLASHI02	-	SDMMC_D0	-
FLASHI03	-	-	-
FLASHI04	-	-	-
FLASHI05	-	-	-
FLASHI06	-	-	-
FLASHI07	-	-	-
FLASHI08	-	-	-
FLASHI09	-	SDMMC_CLK	-
FLASHI010	-	-	-

Boot Source I/O Configuration

The following tables list the boot source I/O settings configured by the boot ROM during initialization. After the preloader is executed, any custom settings you program in Quartus take effect.

Table A-3: QSPI Flash 3.0 V Boot Configuration (BSEL=0x7)

In the "Direction" column for this table, B= bi-directional, I=input, O=output								
Pin	Signal	Configured	Direction	I/O Standard	Drive Strength	Slew Rate Setting	Open Drain	Pull-up
MIXED1I015	QSPI_I00	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I016	QSPI_I01	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I017	QSPI_I02	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I018	QSPI_I03	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I019	QSPI_SS0	Yes	O	3.0-V LVC MOS	4 mA	Fast (1)	No	Yes
MIXED1I020	QSPI_CLK	Yes	O	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I021	QSPI_SS1	No	I	3.0-V LVC MOS	12 mA	Fast (1)	No	Yes

Table A-4: QSPI Flash 1.8 V Boot Configuration (BSEL=0x6)

In the "Direction" column for this table, B= bi-directional, I=input, O=output								
Pin	Signal	Configured	Direction	I/O Standard	Drive Strength	Slew Rate Setting	Open Drain	Pull-up
MIXED1I015	QSPI_I00	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I016	QSPI_I01	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I017	QSPI_I02	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I018	QSPI_I03	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I019	QSPI_SS0	Yes	O	1.8-V LVC MOS	4 mA	Fast (1)	No	Yes
MIXED1I020	QSPI_CLK	Yes	O	1.8-V LVC MOS	4 mA	Fast (1)	No	No

Pin	Signal	Configured	Direction	I/O Standard	Drive Strength	Slew Rate Setting	Open Drain	Pull-up
MIXED1I021	QSPI_SS1	No	I	1.8-V LVCMOS	12 mA	Fast (1)	No	Yes

Table A-5: SD/MMC Internal Transceiver 3.0-V Boot Configuration (BSEL=0x5)

In the "Direction" column for this table, B= bi-directional, I=input, O=output

Pin	Signal	Configured	Direction	I/O Standard	Drive Strength	Slew Rate Setting	Open Drain	Pull-up
FLASHI00	SDMMC_CMD	Yes	B	3.0-V LVCMOS	4 mA	Fast (1)	No	No
FLASHI01	SDMMC_PWREN	No	I	3.0-V LVCMOS	12 mA	Fast (1)	No	Yes
FLASHI02	SDMMC_D0	Yes	B	3.0-V LVCMOS	4 mA	Fast (1)	No	No
FLASHI03	SDMMC_D1	No	I	3.0-V LVCMOS	12 mA	Fast (1)	No	Yes
FLASHI04	SDMMC_D4	No	I	3.0-V LVCMOS	12 mA	Fast (1)	No	Yes
FLASHI05	SDMMC_D5	No	I	3.0-V LVCMOS	12 mA	Fast (1)	No	Yes
FLASHI06	SDMMC_D6	No	I	3.0-V LVCMOS	12 mA	Fast (1)	No	Yes
FLASHI07	SDMMC_D7	No	I	3.0-V LVCMOS	12 mA	Fast (1)	No	Yes
FLASHI08	SDMMC_CLKIN	No	I	3.0-V LVCMOS	12 mA	Fast (1)	No	Yes
FLASHI09	SDMMC_CLK	Yes	O	3.0-V LVCMOS	4 mA	Fast (1)	No	No
FLASHI010	SDMMC_D2	No	I	3.0-V LVCMOS	12 mA	Fast (1)	No	Yes
FLASHI011	SDMMC_D3	No	I	3.0-V LVCMOS	12 mA	fast (1)	No	Yes

Table A-6: SD/MMC Internal Transceiver 1.8-V Boot Configuration (BSEL=0x4)

In the "Direction" column for this table, B= bi-directional, I=input, O=output

Pin	Signal	Configured	Direction	I/O Standard	Drive Strength	Slew Rate Setting	Open Drain	Pull-up
FLASHI00	SDMMC_CMD	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
FLASHI01	SDMMC_PWREN	No	I	1.8-V LVC MOS	12 mA	Fast (1)	No	Yes
FLASHI02	SDMMC_D0	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
FLASHI03	SDMMC_D1	No	I	1.8-V LVC MOS	12 mA	Fast (1)	No	Yes
FLASHI04	SDMMC_D4	No	I	1.8-V LVC MOS	12 mA	Fast (1)	No	Yes
FLASHI05	SDMMC_D5	No	I	1.8-V LVC MOS	12 mA	Fast (1)	No	Yes
FLASHI06	SDMMC_D6	No	I	1.8-V LVC MOS	12 mA	Fast (1)	No	Yes
FLASHI07	SDMMC_D7	No	I	1.8-V LVC MOS	12 mA	Fast (1)	No	Yes
FLASHI08	SDMMC_CLKIN	No	I	1.8-V LVC MOS	12 mA	Fast (1)	No	Yes
FLASHI09	SDMMC_CLK	Yes	O	1.8-V LVC MOS	4 mA	Fast (1)	No	No
FLASHI010	SDMMC_D2	No	I	1.8-V LVC MOS	12 mA	Fast (1)	No	Yes
FLASHI011	SDMMC_D3	No	I	1.8-V LVC MOS	12 mA	Fast (1)	No	Yes

Table A-7: NAND Flash 3.0-V Boot Configuration (BSEL=0x3)

In the "Direction" column for this table, B= bi-directional, I=input, O=output

Pin	Signal	Configured	Direction	I/O Standard	Drive Strength	Slew Rate Setting	Open Drain	Pull-up
MIXED1I00	NAND_ALE	Yes	O	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I01	NAND_CE	Yes	O	3.0-V LVC MOS	4 mA	Fast (1)	No	Yes
MIXED1I02	NAND_CLE	Yes	O	3.0-V LVC MOS	4 mA	fast (1)	No	No
MIXED1I03	NAND_RE	Yes	O	3.0-V LVC MOS	4 mA	Fast (1)	No	Yes

Pin	Signal	Configured	Direction	I/O Standard	Drive Strength	Slew Rate Setting	Open Drain	Pull-up
MIXED1I04	NAND_RB	Yes	I	3.0-V LVC MOS	12 mA	Fast (1)	No	No
MIXED1I05	NAND_DQ0	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I06	NAND_DQ1	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I07	NAND_DQ2	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I08	NAND_DQ3	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I009	NAND_DQ4	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I010	NAND_DQ5	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I011	NAND_DQ6	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I012	NAND_DQ7	Yes	B	3.0-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I013	NAND_WP	Yes	O	3.0-V LVC MOS	4 mA	Fast (1)	No	Yes
MIXED1I014	NAND_WE	Yes	O	3.0-V LVC MOS	4 mA	Fast (1)	No	Yes

Table A-8: NAND Flash 1.8-V Boot Configuration (BSEL=0x2)

In the "Direction" column for this table, B= bi-directional, I=input, O=output

Pin	Signal	Configured	Direction	I/O Standard	Drive Strength	Slew Rate	Open Drain	Pull-up
MIXED1I00	NAND_ALE	Yes	O	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I01	NAND_CE	Yes	O	1.8-V LVC MOS	4 mA	Fast (1)	No	Yes
MIXED1I02	NAND_CLE	Yes	O	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I03	NAND_RE	Yes	O	1.8-V LVC MOS	4 mA	Fast (1)	No	Yes
MIXED1I04	NAND_RB	Yes	I	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I05	NAND_DQ0	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No

Pin	Signal	Configured	Direction	I/O Standard	Drive Strength	Slew Rate	Open Drain	Pull-up
MIXED1I06	NAND_DQ1	Yes	B	1.8-V LVC MOS	12 mA	Fast (1)	No	No
MIXED1I07	NAND_DQ2	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I08	NAND_DQ3	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I09	NAND_DQ4	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I010	NAND_DQ5	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I011	NAND_DQ6	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I012	NAND_DQ7	Yes	B	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I013	NAND_WP	Yes	O	1.8-V LVC MOS	4 mA	Fast (1)	No	No
MIXED1I014	NAND_WE	Yes	O	1.8-V LVC MOS	4 mA	Fast (1)	No	No

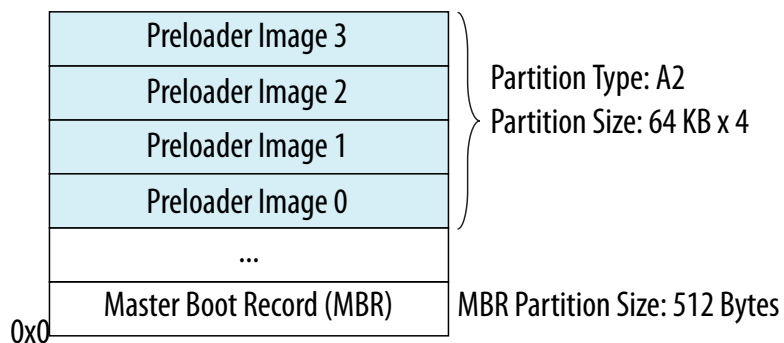
Flash Memory Devices for Booting

The memory controllers and devices that contain the boot loader image have configuration requirements for proper boot from flash.

SD/MMC Flash Devices

The following figure shows an SD/MMC flash image layout example for boot. The master boot record (MBR) is located in the first 512 bytes of the memory. The MBR contains information about the partitions (address and size of partition). The preloader image is stored in partition A2. Partition A2 is a custom raw partition with no file system.

Figure A-5: SD/MMC Flash Image Layout



The SD/MMC controller supports two booting modes:

- MBR (partition) mode
 - The boot image is read from a custom partition (0xA2)
 - The first image is located at the beginning of the partition, at offset 0x0
 - Start address = partition start address
- Raw mode
 - If the MBR signature is not found, SD/MMC driver assumes it is in raw mode
 - The boot image data is read directly from sectors in the user area and is located at the first sector of the SD/MMC
 - The first image is located at the start of the memory card, at offset 0x0
 - Start address = 0x0

The MBR contains the partition table, which is always located in the first sector (LBA0) with a memory size of 512 bytes. The MBR consists of executable code, four partition entries, and the MBR signature. A MBR can be created by specific tools like the FDISK program.

Table A-9: MBR Structure

Offset	Size (In Bytes)	Description
0x000	446	Code area
0x1BE	16	Partition entry for partition 1
0x1CE	16	Partition entry for partition 2
0x1DE	16	Partition entry for partition 3
0x1EE	16	Partition entry for partition 4
0x1FE	2	MBR signature: 0xAA55

The standard MBR structure contains a partition with four 16-byte entries. Thus, memory cards using this standard table cannot have more than four primary partitions or up to three primary partitions and one extended partition.

Each partition type is defined by the partition entry. The boot images are stored in a primary partition with custom partition type (0xA2). The SD/MMC flash driver does not support a file system, so the boot images are located in partition A2 at fixed locations.

Table A-10: Partition Entry

Offset	Size (In Bytes)	Description
0x0	1	Boot indicator. 0x80 indicates that it is bootable.
0x1	3	Starting CHS value
0x4	1	Partition type
0x5	3	Ending CHS value
0x8	4	LBA of first sector in partition
0xB	4	Number of sectors in partition

The boot ROM code configures the SD/MMC controller to default settings for the supported SD/MMC flash memory.

Related Information

[SD/MMC Controller](#) on page 14-1

Refer to the *SD/MMC Controller* chapter for more information regarding features and functionality.

Default Settings of the SD/MMC Controller

Table A-11: SD/MMC Controller Default Settings

Parameter	Default	Register Value
Card type	1 bit	The card type register (ctype) in the SD/MMC controller registers (sdmmc) = 0x0
Bus mode	—	SD/MMC ⁽⁶⁶⁾
Timeout	Maximum	The timeout register (tmout) = 0xFFFFFFFF
FIFO threshold RX watermark level	1	The RX watermark level field (rx_wmark) of the FIFO threshold watermark register (fifoth) = 0x1

⁽⁶⁶⁾ SPI cards are not supported for boot.

Parameter		Default	Register Value
Clock source		0	The clock source register (clksrc) = 0x0
Block size		512	The block size register (blksiz) = 0x200
Clock divider	Identification mode	32	The clock divider register (clkdiv) = 0x10 ($2 \times 16 = 32$)
	Data transfer mode	Bypass	The clock divider register (clkdiv) = 0x00
External Device Power enable		Disabled (Power Off)	The power_enable bit in the pwren register is programmed to 0x0 out of reset, meaning the sdmmc_pwren signal is not active and does not do anything at boot time. A pull-up should be attached to the sdmmc_pwren pin for proper functionality. Note: If the SD/MMC controller is used as the boot source, then power to the corresponding SD card must be supplied from a power controller on the board. After boot has completed and the SD/MMC controller has been fully configured, the sdmmc_pwren signal can be used to turn on and off the SD card through the pwren register.

CSEL Settings for the SD/MMC Controller

Table A-12: SD/MMC Controller CSEL Pin Settings

Setting		CSEL[1:0] Pin Value			
		0	1	2	3
osc1_clk (HPS1_CLK pin) range		10–50 MHz	10–12.5 MHz	12.5–25 MHz	25–50 MHz
ID mode	Device clock (sdmmc_clk_out)	osc1_clk/512, 97.66 KHz max	osc1_clk/128, 97.66 KHz max	osc1_clk/256, 97.66 KHz max	osc1_clk/512, 97.66 KHz max
	Controller baud rate divisor	32	32	32	32
Data transfer mode	Device clock (sdmmc_clk_out)	osc1_clk/4, 12.5 MHz max	osc1_clk*1, 12.5 MHz max	osc1_clk/2, 12.5 MHz max	osc1_clk/4, 12.5 MHz max
	Controller baud rate divisor (even numbers only)	1 (bypass)	1 (bypass)	1 (bypass)	1 (bypass)
Controller clock (sdmmc_clk)		osc1_clk, 50 MHz max	osc1_clk, 50 MHz max	osc1_clk, 50 MHz max	osc1_clk, 50 MHz max
mpu_clk		osc1_clk, 50 MHz max	osc1_clk*32, 400 MHz max	osc1_clk*16, 400 MHz max	osc1_clk*8, 400 MHz max
PLL modes		Bypassed	Locked	Locked	Locked

NAND Flash Devices

The NAND subsystem reserves at least the first 256 KB on the NAND device. If the NAND flash device has blocks greater than 64 KB, then the NAND subsystem reserves the first four blocks on the device. For a NAND device with less than 64 KB block size, the preloader image must be placed in multiple blocks. The NAND subsystem expects to find up to four preloader images on the NAND device. You may have less than four if required. The preloader image should always be at the start of a physical page. Because a block is the smallest area used for erase operation, any update to a particular image does not affect other images.

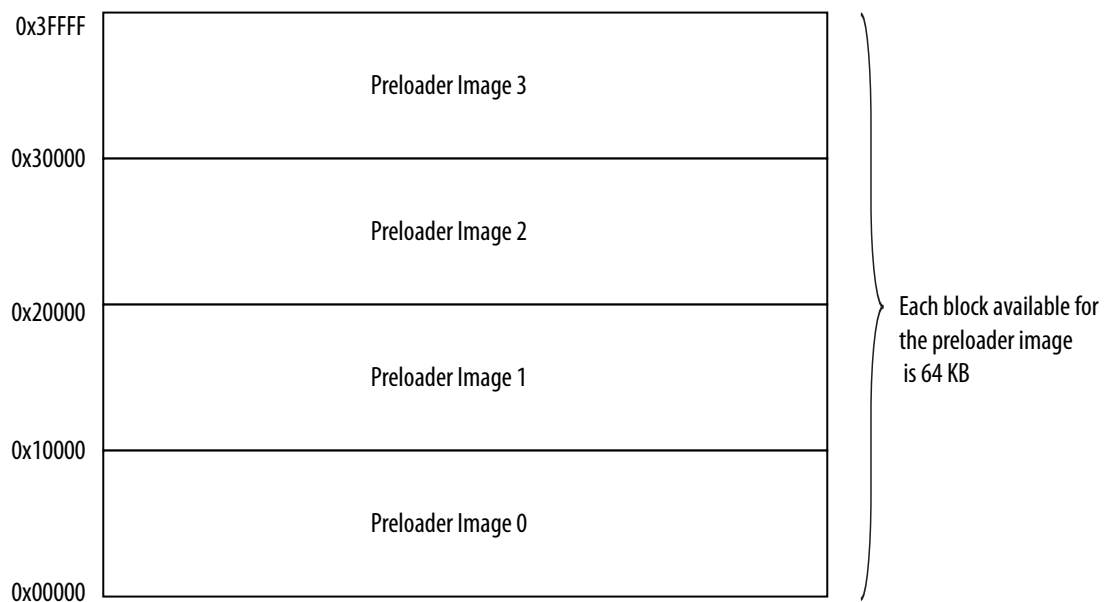
Figure A-6: NAND Flash Image Layout for 64 KB Memory Blocks

Figure A-7: NAND Flash Image Layout for 128 KB Memory Blocks

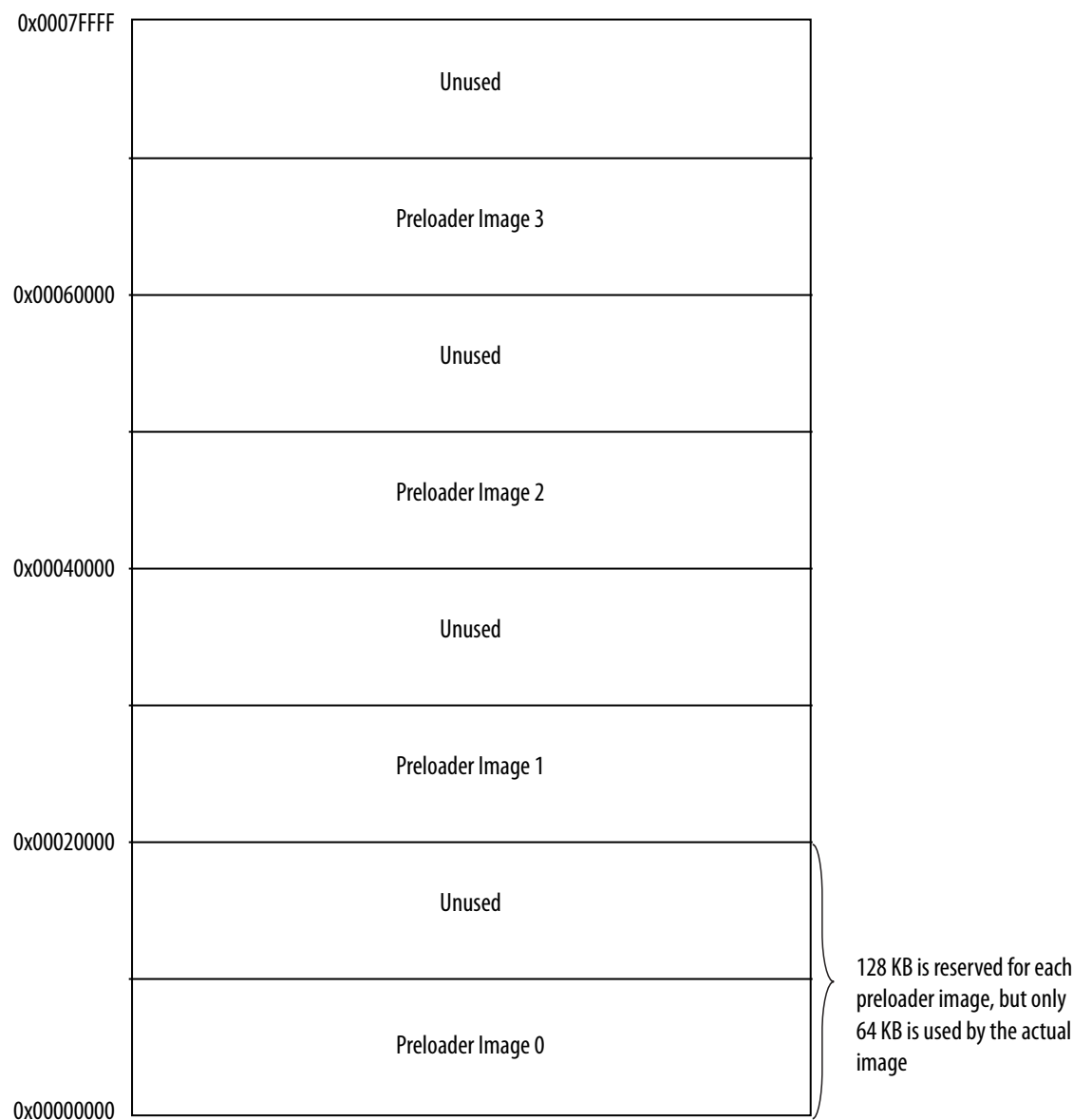
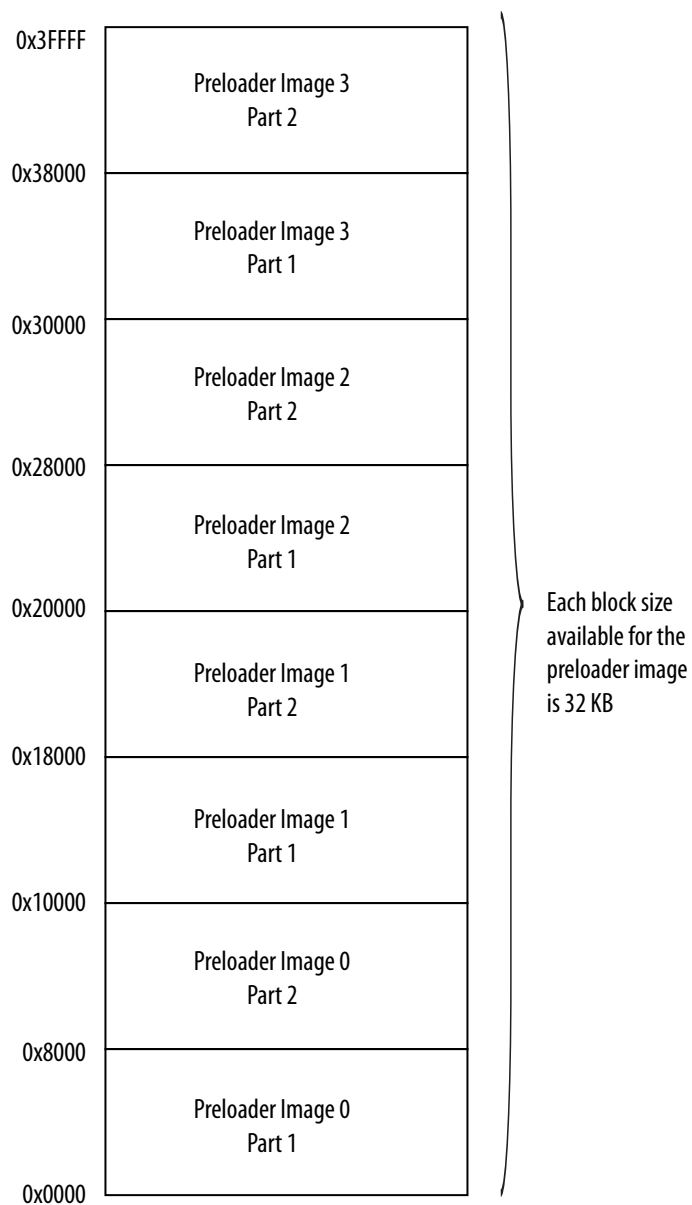


Figure A-8: NAND Flash Image Layout for 32 KB Memory Blocks

**Related Information**

[NAND Flash Controller](#) on page 13-1

Refer to the *NAND Flash Controller* chapter for more information regarding features and functionality.

NAND Flash Driver Features Supported in the Boot ROM Code

Table A-13: NAND Flash Support Features

Feature	Driver Support
Device	Open NAND Flash Interface (ONFI) 1.0 raw NAND or electronic signature devices, single layer cell (SLC) and multiple layer cell (MLC)
Chip select	CS0 only. Only CS0 is available to the HPS, the other three chip selects are routed out to the FPGA portion of the device
Bus width	8-bit only
Page size	512 bytes, 2 KB, 4 KB, or 8 KB
Page per block	32, 64, 128, 384 and 512
ECC	512-bytes with 8-bit correction

CSEL Settings for the NAND Controller

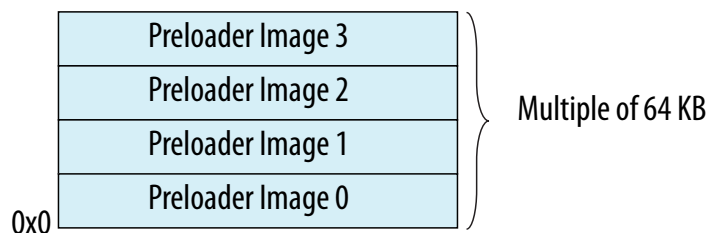
Table A-14: NAND Controller CSEL Pin Settings

Setting	CSEL[1:0] Pin Value			
	0	1	2	3
osc1_clk (HPS1_CLK pin) range	10–50 MHz	10–12.5 MHz	12.5–25 MHz	25–50 MHz
Device frequency (nand_x_clk/4)	osc1_clk/4, 12.5 MHz max	osc1_clk*8/4, 25 MHz max	osc1_clk*4/4, 25 MHz max	osc1_clk2/4, 25 MHz max
Controller clock (nand_x_clk)	osc1_clk, 50 MHz max	osc1_clk*8, 100 MHz max	osc1_clk*4, 100 MHz max	osc1_clk*2, 100 MHz max
mpu_clk	osc1_clk, 50 MHz max	osc1_clk*32, 400 MHz max	osc1_clk*16, 400 MHz max	osc1_clk*8, 400 MHz max
PLL modes	Bypassed	Locked	Locked	Locked

Quad SPI Flash Devices

The figure below shows the quad SPI flash image layout. The preloader image is always located at offsets that are multiples of 64 KB.

Figure A-9: Quad SPI Flash Image Layout



The boot ROM code configures the quad SPI controller to default settings for the supported SPI or quad SPI flash memory.

Related Information

[Quad SPI Flash Controller](#) on page 15-1

Refer to the *Quad SPI Flash Controller* chapter for more information regarding features and functionality.

Quad SPI Controller Default Settings

Table A-15: Quad SPI Controller Default Settings

Parameter	Default Setting	Register Value
SPI baud rate	Divide by 4	The master mode baud rate divisor field (bauddiv) of the quad SPI configuration register (cfg) in the quad SPI controller registers (qspiregs) = 1.
Read opcode	CSEL = 0 or 1: Normal read CSEL = 2 or 3: Fast read	The read opcode in non-XIP mode field (rdopcode) in the device read instruction register (devrd) = 0x3 (for normal read) and 0xB (for fast read).
Instruction type	Single I/O (1 bit wide)	The address transfer width field (addrwidth) and data transfer width field (datawidth) of the devrd register = 0.
Number of address bytes	3 bytes	The number of address bytes field (numaddrbytes) of the device size register (devsz) = 2. Note: Before a reset, you must ensure that the QSPI flash device is configured to 3 byte address mode for the boot ROM to function properly.

Parameter	Default Setting	Register Value
Delay in terms of <code>qspi_ref_clk</code> clock cycles for the length that the master mode chip select outputs are deasserted between words when the clock phase is zero	The default setting in clock cycles must equal 200 ns	The clock delay for chip select deassert (field <code>nss</code> in the quad SPI device delay register (<code>delay</code>)). Refer to <code>delay[31:24]</code> in the "Quad SPI Flash Delay Configuration" table in the "Quad SPI Flash Delay Configuration" section for calculations.
Delay in terms of <code>qspi_ref_clk</code> clock cycles between one chip select being deactivated and the activation of another. This delay ensures a quiet period between the selection of two different slaves and requires the transmit FIFO to be empty	The default setting is 0 clock cycles.	The clock delay for chip select deactivation (field <code>btwn</code> in the <code>delay</code> register = 0x0).
Delay in terms of <code>qspi_ref_clk</code> clock cycles between the last bit of the current transaction and the first bit of the next transaction. If the clock phase is zero, the first bit of the next transaction refers to the cycle in which the chip select is deselected	The default setting in clock cycles must equal 20 ns.	The clock delay for last transaction bit (field <code>after</code> in the <code>delay</code> register). Refer to <code>delay[15:8]</code> in the "Quad SPI Flash Delay Configuration" table in the "Quad SPI Flash Delay Configuration" section for calculations.
Added delay in terms of <code>qspi_ref_clk</code> clock cycles between setting <code>qspi_nss_out</code> low and first bit transfer	The default setting in clock cycles must equal 20 ns.	The clock delay of <code>qspi_nss_out</code> (field <code>init</code> in the <code>delay</code> register). Refer to <code>delay[7:0]</code> in the "Quad SPI Flash Delay Configuration" table in the "Quad SPI Flash Delay Configuration" section for calculations.

Quad SPI Flash Delay Configuration

The `delay` register in the quad SPI controller configures relative delay of the generation of the master output signals. All timings are defined in cycles of `qspi_ref_clk`.

The quad SPI flash memory must meet the following timing requirements:

- T_{SLCH} : 20 ns
 - T_{SLCH} is used to calculate the `init` field (`delay[7:0]`) in the `delay` register. The `init` field represents the delay in `qspi_ref_clk` clocks between pulling the device chip select (`qspi_n_ss_out`) low and the first bit transfer.
- T_{CHSH} : 20 ns
 - T_{CHSH} is used to calculate the `after` field (`delay[15:8]`) in the `delay` register. The `after` field represents the delay in the `qspi_ref_clk` clocks between last bit of the current transaction and the deassertion of the device chip select (`qspi_n_ss_out`).
- T_{SHSL} : 200 ns
 - T_{SHSL} is used to calculate the `nss` field (`delay[31:24]`) in the `delay` register and is the delay in the `qspi_ref_clk` clocks for the length that the master mode chip select outputs are deasserted between transactions.
- $T_{qspi\_ref\_clk}$ is the master reference clock/external clock, `qspi_ref_clk`.

The formulas to calculate the fields in the `delay` register are:

$$\text{delay}[7:0] = \text{init} = T_{SLCH} / T_{qspi\_ref\_clk}$$

$$\text{delay}[15:8] = \text{after} = T_{CHSH} / T_{qspi\_ref\_clk}$$

$$\text{delay}[31:24] = \text{nss} = (T_{SHSL} - T_{qspi\_clk}) / T_{qspi\_ref\_clk}$$

Table A-16: Quad SPI Flash Delay Configuration

CSEL[1:0] Pin Value	$T_{qspi\_ref\_clk}$ (ns)	T_{qspi_clk} (ns)	Device Delay Register		
			<code>delay[7:0] (init)</code>	<code>delay[15:8] (after)</code>	<code>delay[31:24] (nss)</code>
0	20	80	1	1	6
1	10	40	2	2	16
2–3	5	20	4	4	36

Read data capture delay is also configured when booting from QSPI. Depending on the CSEL pin settings, the Boot ROM configures the `delay` field of the `rddatacap` register in the QSPI differently. When `CSEL Pins[1:0] = 0x0` or `0x1`, the Boot ROM leaves the `delay` field in the `rddatacap` register untouched. When `CSEL Pins[1:0] = 0x2` or `0x3`, the boot ROM calibrates the interface by reading the QSPI signal for all delay values of the `rddatacap` register. The Boot ROM analyzes all of the delay values that return a valid signature and uses the delay value in the middle of the valid window as the value it programs into the `delay` field of the `rddatacap` register.

Quad SPI Controller CSEL Settings

Table A-17: Quad SPI Controller CSEL Pin Settings

Setting	CSEL[1:0] Pin Value			
	0 ⁽⁶⁷⁾	1	2	3
osc1_clk (HPS1_CLK pin) range	10–50 MHz	20–50 MHz	25–50 MHz	10–25 MHz
Device clock (qspi_clk)	osc1_clk/4, 12.5 MHz max	osc1_clk/2, 25 MHz max	osc1_clk*1, 50 MHz max	osc1_clk*2, 50 MHz max
Controller clock (qspi_ref_clk)	osc1_clk, 50 MHz max	osc1_clk*2, 100 MHz max	osc1_clk*4, 200 MHz max	osc1_clk*8, 200 MHz max
Controller baud rate divisor (even numbers only)	4	4	4	4
Flash read instruction (1 dummy byte for READ_FAST)	READ	READ	READ_FAST	READ_FAST
mpu_clk	osc1_clk, 50 MHz max	osc1_clk*8, 400 MHz max	osc1_clk*8, 400 MHz max	osc1_clk*16, 400 MHz max
PLL modes	Bypassed	Locked	Locked	Locked

Clock Select

The boot ROM reads the clock select values to determine what frequency has been selected for the CPU clock and any interface clock during boot.

The clock select (CSEL) pins are asserted to select the speed of the HPS boot interface. If the FPGA is used as the boot source, the CSEL pins are ignored. The CSEL values define the main PLL, l2_mp_clk and l4_sp_clk. Based on the clock source and clock select settings, boot ROM configures the main PLL and peripheral PLL parameters and the clock dividers for clocks derived from the PLLs.

Note: The terms CSEL and CLKSEL are used interchangeably in Intel documentation to refer to clock select.

Note: At power-up or reset, when CSEL[1:0]=0x0 or BSEL[2:0] is configured to boot from FPGA, the HPS PLLs are in bypass mode. Thus, HPS user clocks exported to the FPGA fabric run at osc1_clk frequency.

I/O Configuration

The flash devices needed for booting must be connected to specific I/Os. The Boot ROM configures these I/Os depending on the flash device selected by boot select setting. To configure the I/Os, the boot ROM performs pin muxing and pin configuration on these I/Os.

At power up, the dedicated I/O are in tri-state with a weak pull-up until the Boot ROM or preloader configures the I/O. On cold reset, the boot ROM always configures the pin muxes and pin configurations.

⁽⁶⁷⁾ Not applicable when on WARM reset.

On warm reset, the user has the capability to specify whether or not the boot ROM code configures the I/Os and pin muxes for boot pins after a warm reset. This configuration on warm reset is enabled by programming the `ctrl` register in the Boot ROM Code Register group of the System Manager.

L4 Watchdog 0 Timer

The boot ROM enables the L4 Watchdog 0 Timer early in the boot process.

This watchdog is reserved for the boot ROM until the preloader indicates that it has started correctly and taken control of the exception vectors. The timeout is at least one second, depending on the clock select setting. Because the watchdog is reset just before the control passes to preloader, the preloader must reset the watchdog when it begins execution.

The L4 watchdog 0 timer is reserved for boot ROM use. While booting, if a watchdog reset happens before software control passes to the preloader, the boot ROM code attempts to load the last valid preloader image, identified by the `initswlastld` register in the System Manager.

If the watchdog reset happens after the preloader has started executing but before it writes a valid value to `initswstate` register, the boot ROM increments `initswlastld` and attempts to load that image. If the watchdog reset happens after the preloader writes a valid value to `initswstate` register, the boot ROM code attempts to load the image indicated by `initswlastld` register.

Preloader

The function of the preloader is user-defined. However, typical functions include:

- Initializing the SDRAM interface
- Configuring the HPS I/O pins
- Initializing the interface that loads the next stage of software

Initializing the SDRAM allows the preloader to load the next stage of the boot software (that might not fit in the 60 KB available in the on-chip RAM) into SDRAM. A typical next software stage is the open source boot loader, U-boot. The preloader is allowed to load the next boot software stage from any device available to the HPS. Typical sources include the same flash device that contains the preloader, a different flash device, or a communication interface such as an EMAC.

U-Boot Loader

The optional U-boot loader loads the operating system and passes software control to the operating system.

Boot ROM Flow

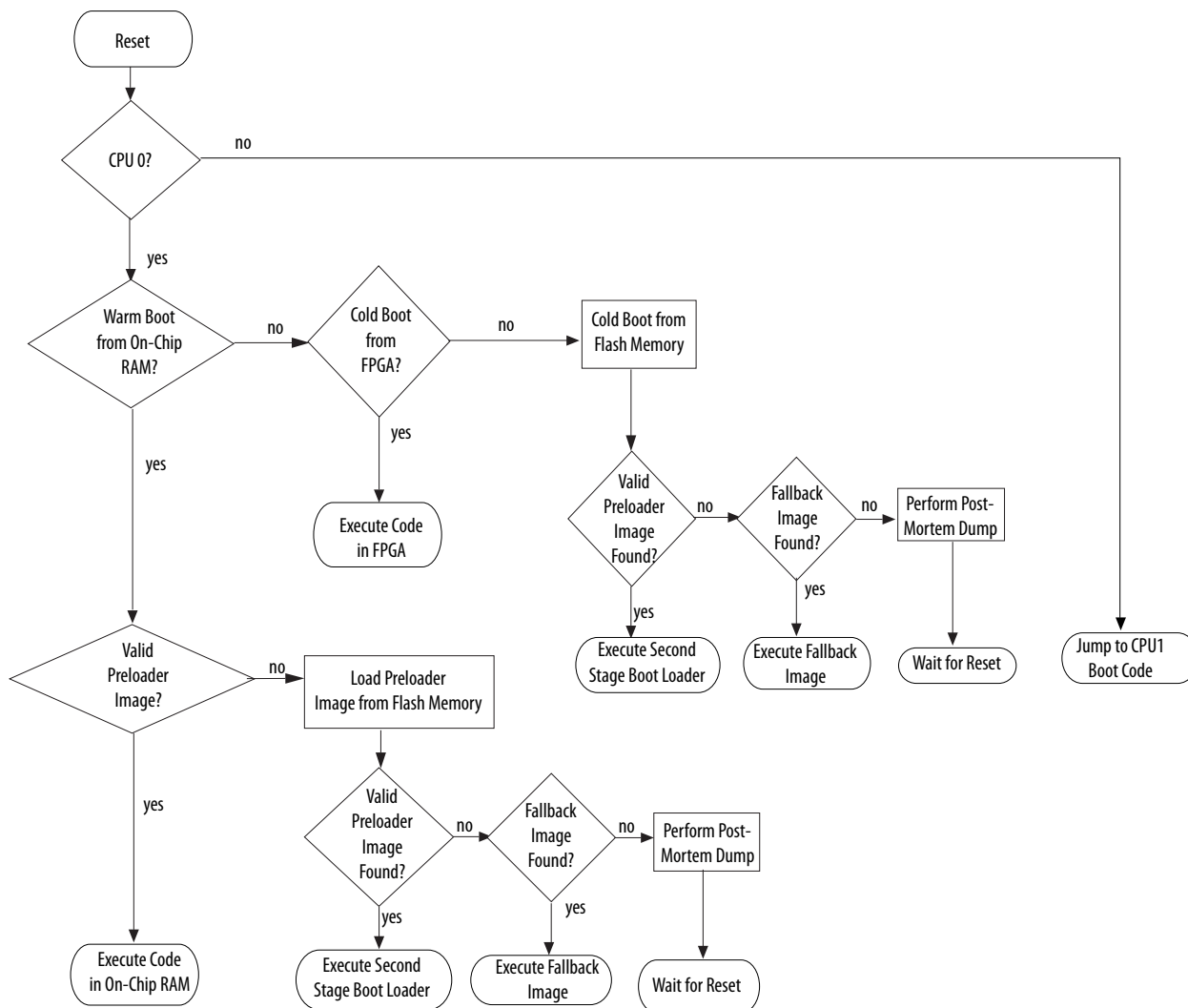
On a cold reset, the HPS boot process starts when CPU0 is released from reset (for example, on a power up) and executes code in the internal boot ROM at the reset exception address. The boot ROM code brings the SoC out of reset and into a known state. After boot ROM code is exited, control passes to the next stage of the boot software, referred to as the preloader. The preloader can be customized and is typically stored external to the HPS in a nonvolatile flash-based memory or in on-chip RAM within the FPGA. Beyond that, another boot layer can be executed before loading the operating system software.

This section describes the software flow from reset until the boot ROM code passes software control to the preloader.



The following figure illustrates that the boot ROM code can perform a warm boot from on-chip RAM, a cold boot from the FPGA portion of the device, or a cold boot from flash memory.

Figure A-10: BOOT ROM FLOW



After a reset, the boot ROM code determines which CPU it is executing on. If it is executing on CPU 1, then control is passed to CPU1 and the boot ROM execution is complete. This event only happens if the user code releases CPU1 without resetting the exception vectors.

If the boot ROM is executing on CPU0, it checks to see if an on-chip RAM boot is requested. An on-chip RAM boot can only occur on a warm reset. Warm boot from on-chip RAM has the highest priority to execute if the warmramgrp registers in the romcodegrp group in the system manager has been configured to support booting from on-chip RAM on a warm reset. If the enable register in the System Manager is set to 0xAE9EFEB, then the boot ROM code attempts to boot from on-chip RAM on a warm reset and the boot ROM does not configure boot I/Os, pin-muxes or clocks. The datastart and length registers in System Manager allow you to program the offset of the beginning of code and the length of the region of

on-chip RAM for CRC validation. If the `length` register is clear, then boot ROM does not perform a CRC calculation on the on-chip RAM.

If a valid preloader image cannot be found in the on-chip RAM, or the preloader in the on-chip RAM fails the CRC check, the boot ROM code attempts to load the last valid preloader image loaded from the flash memory, identified by the `index` field of the initial software last image loaded register (`initswlastld`) in the `romcodegrp` group in the system manager. If the image is invalid, boot ROM code attempts to load up to three subsequent images from flash memory. If a valid preloader image cannot be found in the on-chip RAM or flash memory, the boot ROM code checks the FPGA portion of the device for a fallback image.

If the warm RAM boot has failed or if a cold reset has occurred, then the boot ROM reads the BSEL value in the `bootinfo` register of the System Manager. If the FPGA is selected as the boot source, then the boot ROM code attempts to execute code at address `0xC0000000` across the HPS-to-FPGA bridge (offset `0x00000000` from bridge). No error conditions are generated if the FPGA does not initialize properly and the watchdog is not enabled for time-out. Instead, the boot ROM continues to wait until the FPGA is available.

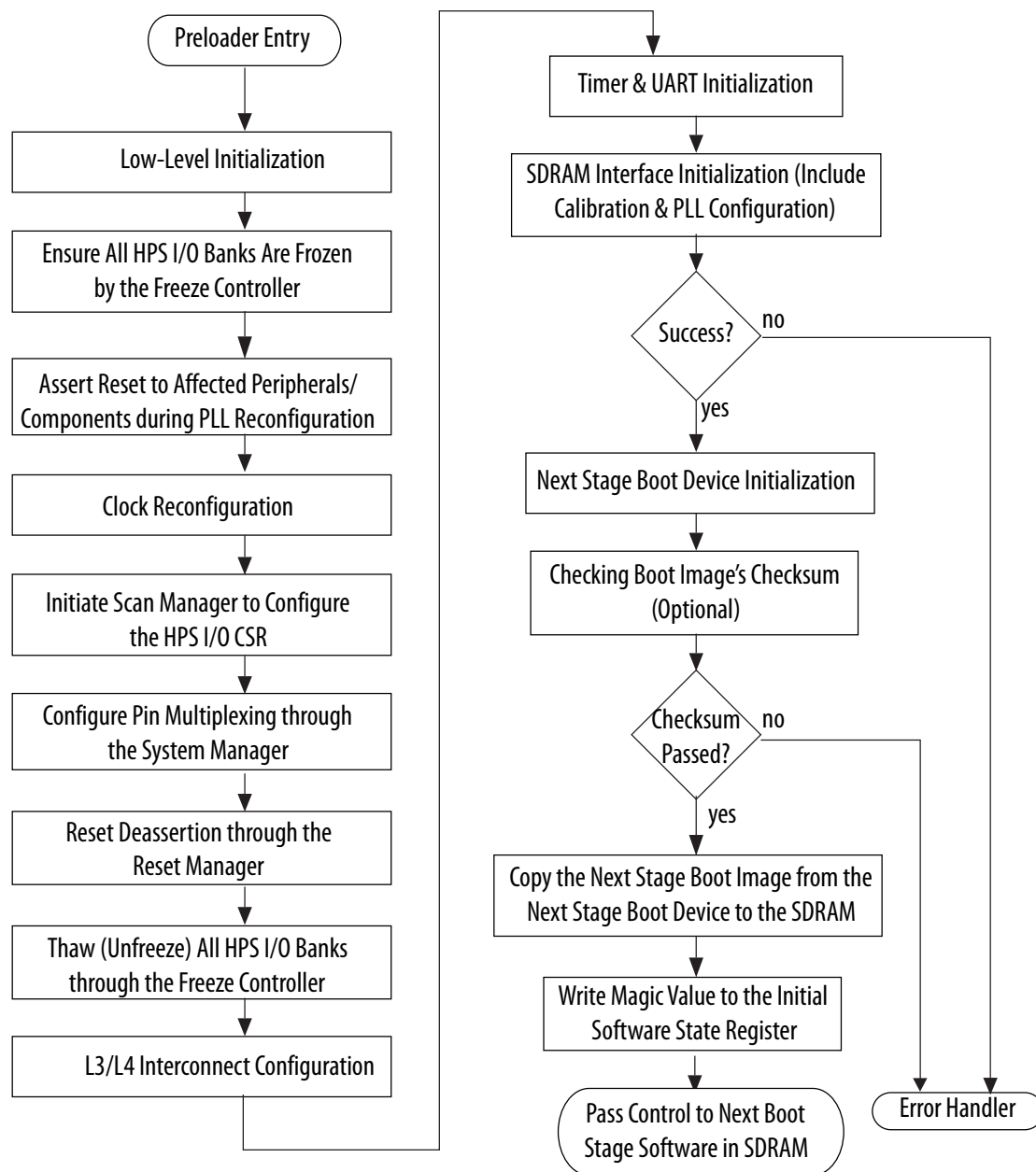
If the BSEL bits indicate a boot from external flash, then the boot ROM code attempts to load an image from a flash device into the on-chip RAM, verify and execute it. If the BSEL is invalid or the boot ROM code cannot find a valid image in the flash, then the boot ROM code checks if there is a fallback image in the FPGA. If there is, then the boot ROM executes the fallback image. If there is no fallback image then the boot ROM performs a post-mortem dump of information into the on-chip RAM and awaits a reset.

Note: The acronyms BSEL and BOOTSEL are used interchangeably to define the boot select pins.

Typical Preloader Boot Flow

This section describes a typical software flow from the preloader entry point until the software passes control to the next stage boot software.

Figure A-11: Typical Preloader Flow



Low-level initialization steps include reconfiguring or disabling the L4 watchdog 0 timer, invalidating the instruction cache and branch predictor, remapping the on-chip RAM to the lowest memory region, and setting up the data area.

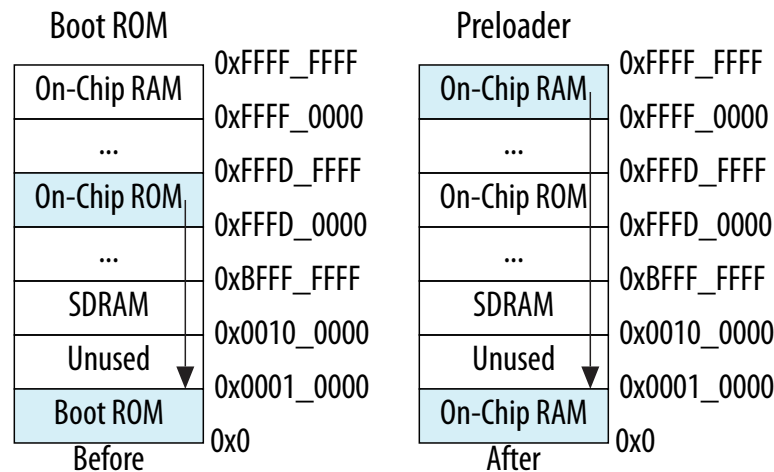
Upon entering the preloader, the L4 watchdog 0 timer is active. The preloader can disable, reconfigure, or leave the watchdog timer unchanged. Once enabled after reset, the watchdog timer cannot be disabled, only paused.

The instruction cache and branch predictor, which were previously enabled by the boot ROM code, need to be invalidated.

The preloader needs to remap the exception vector table because the exception vectors are still pointing to the exception handler in the boot ROM when the preloader starts executing. By setting the L3 interconnect remap bit 0 to high, the on-chip RAM mirrors to the lowest region of the memory map. After this remap, the exception vectors use the exception handlers in the preloader image.

The figure below shows the memory map before and after remap.

Figure A-12: Remapping the On-Chip RAM



The preloader can reconfigure all HPS clocks. During clock reconfiguration, the preloader asserts reset to the peripherals in the HPS affected by the clock changes.

The preloader configures HPS I/O pins through the scan manager and pin multiplexing through the system manager. The preloader initiates the freeze controller in the scan manager to freeze all the I/O pins and put them in a safe state during I/O configuration and pin multiplexing.

Note: HPS handoff files are created in the `hps_isw_handoff` folder during design compilation using Intel Quartus Prime. The handoff files contain pin multiplexing configuration and I/O settings for HPS pins, and are used to generate the required boot loader device tree for proper FPGA hardware initialization and run-time access.

The SDRAM goes through full initialization for cold boot or a partial initialization for warm boot. For full initialization, the preloader configures the SDRAM PLL before releasing the SDRAM interface from reset. SDRAM calibration adjusts I/O delays and FIFO settings to compensate for any board skew or impairment in the board, FPGA portion of the device, or memory device. For partial initialization, SDRAM PLL configuration and SDRAM calibration is not necessary.

The preloader looks for a valid next stage boot image in the next stage boot device by checking the boot image validation data and checksum in the mirror image. Once validated, the preloader copies the next stage boot image from the next stage boot device to the SDRAM.

Before software passes control to the next stage boot software, the preloader can write a valid value (0x49535756) to be read by the preloader in the `initswstate` register under the `romcodegrp` group in the system manager. This value indicates that there is a valid boot image in the on-chip RAM. The `initisw-last1d` register holds the index of the last preloader software image loaded by the Boot ROM from the

boot device. When a warm reset occurs, the Boot ROM loads the image indicated by the `initswlastld` register if the BSEL value is the same as the last boot.

HPS State on Entry to the Preloader

When the boot ROM code is ready to pass control to the preloader, the processor (CPU0) is in the following state:

- Instruction cache is enabled
- Branch predictor is enabled
- Data cache is disabled
- MMU is disabled
- Floating point unit is enabled
- NEON vector unit is enabled
- Processor is in Arm secure supervisor mode

The boot ROM code sets the Arm Cortex-A9 MPCore registers to the following values:

- `r0`—contains the pointer to the shared memory block, which is used to pass information from the boot ROM code to the preloader. The shared memory block is located in the top 4 KB of on-chip RAM.
- `r1`—contains the length of the shared memory.
- `r2`—unused and set to `0x0`.
- `r3`—reserved.

All other MPCore registers are undefined.

Note: When booting CPU0 using the FPGA boot, or when booting CPU1 using any boot source, all MPCore registers, caches, the MMU, the floating point unit, and the NEON vector unit are undefined. HPS subsystems and the PLLs are undefined.

When the boot ROM code passes control to the preloader, the following conditions also exist:

- The boot ROM is still mapped to address `0x0`.
- The L4 watchdog 0 timer is active and has been toggled.

The boot ROM also saves the state of the reset cause in the `stat` register of the Reset Manager and this information is available for the preloader to read.

Shared Memory

The shared memory contains information that the boot ROM code passes to the preloader. The boot ROM code passes the location of shared memory to the preloader in register `r0`, as described in the *HPS State on Entry to Preloader* section.

The shared memory holds the last value of the `stat` register in the Reset Manager Module and a version number that is passed by the boot ROM. The value of the `stat` register in shared memory is only valid if the version number passed by the boot ROM is `0x00000000`. The table below shows where the `stat` register value and version number reside in the shared memory as an offset from the address stored in register `r0`.



Table A-18: Shared Memory Locations

Shared Memory Content	Address Location	Description
Version Number	$r0 + 0x0008$	The stat register value in shared memory is only valid if the value of this shared memory location is 0x00000000.
stat Register	$r0 + 0x0038$	This location holds the last value the boot ROM writes to the stat register in the Reset Manager before it is erased.

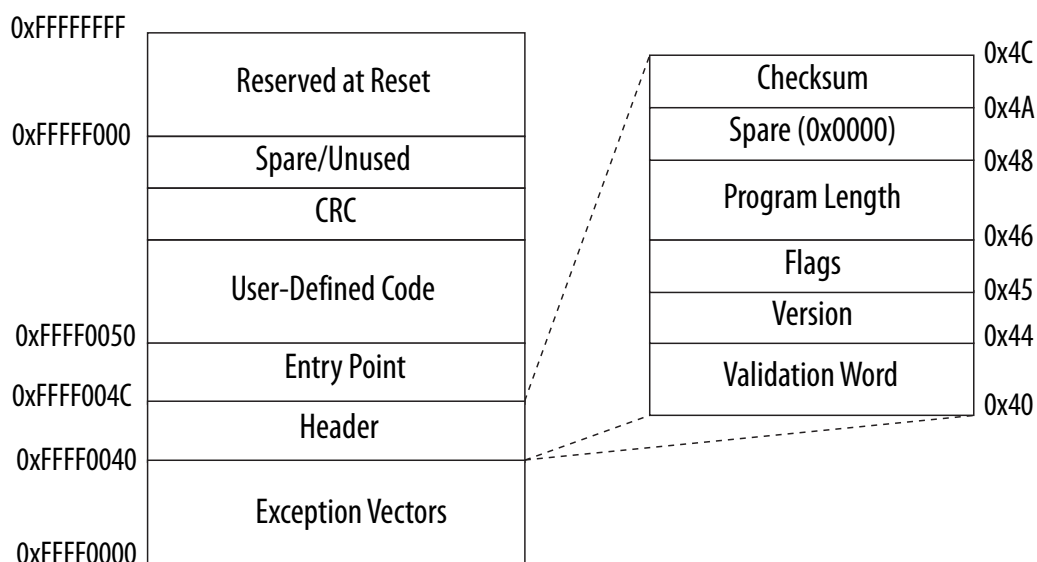
Loading the Preloader Image

The boot ROM code loads the image from flash memory into the on-chip RAM and passes control to the preloader. The boot ROM code checks for a valid image by verifying the header and cyclic redundancy check (CRC) in the preloader image.

The boot ROM code checks the header for the following information:

- Validation word—validates the preloader image. The validation word has a fixed value of 0x31305341.
- Version—indicates the header version.
- Program length—the total length of the image (in 32-bit words) from offset 0x0 to the end of code area, including exception vectors and CRC.
- Checksum—a checksum of all the bytes in the header, from offset 0x40 to 0x49.

The preloader image has a maximum size of 60 KB. This size is limited by the on-chip RAM size of 64 KB, where 4 KB is reserved as a workspace for the boot ROM data and stack. The preloader can use this 4 KB region (for its stack and data, for example) after the boot ROM code passes control to the preloader. This 4 KB region is overwritten by the boot ROM code on a subsequent reset. The following figure shows the preloader image layout in the on-chip RAM after being loaded from the boot ROM.

Figure A-13: Preloader Image Layout

Exception vectors—Exception vectors are located at the start of the on-chip RAM. Typically, the second-stage boot loader remaps the lowest region of the memory map to the on-chip RAM (from the boot ROM) to create easier access to the exception vectors.

Header—contains information such as validation word, version, flags, program length, and checksum for the boot ROM code to validate the preloader boot loader image before passing control to the second-stage boot loader.

Entry point—After the boot ROM code validates the header, the boot ROM code jumps to this address.

User-defined code—typically contains the program code of the preloader.

CRC—contains a CRC of data from address 0xFFFF0000 to 0xFFFF0000+(Program Length*4)–0x0004. The polynomial used to validate the preloader image is:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

There is no reflection of the bits. The initial value of the remainder is 0xFFFFFFFF and the final value is XORed with 0xFFFFFFFF.

Reserved at reset—the top 4 KB is reserved for the boot ROM code after a reset. The boot ROM code uses this area for internal structures, workspace, and post-mortem dump. This area includes the shared memory where the boot ROM code passes information to the preloader.

FPGA Configuration

You can configure the FPGA portion of the SoC device with non-HPS sources or by utilizing the HPS.

When the HPS handles the initial FPGA configuration, software executing on the HPS writes the configuration image to the FPGA Manager in the HPS. Software can control the configuration process and monitor the FPGA status by accessing the control and status register (CSR) interface in the FPGA Manager.

Related Information

- [FPGA Manager](#) on page 5-1
For more information regarding programming the FPGA through the HPS, refer to the *FPGA Manager* chapter.
- [Configuration, Design Security, and Remote System Upgrades in Arria V Devices](#)

Full Configuration

The HPS uses the FPGA manager to configure the FPGA portion of the device. The following sequence suggests one way for software to perform a full configuration:

- CONF_DONE = 1 and nSTATUS = 1 indicates successful configuration.
- CONF_DONE = 0 or nSTATUS = 0 indicates unsuccessful configuration. Complete steps 12 and 13, then go back and repeat steps 3 to 10 to reload the configuration image.
- If DCLK is unused, write a value of 4 to the DCLK count register (dclkcnt).
- If DCLK is used, write a value of 20,480 (0x5000) to the dclkcnt register.

If the HPS resets in the middle of a normal configuration data transfer before entering user mode, software can assume that the configuration is unsuccessful. After the HPS resets, software must repeat the steps for full configuration.

1. Set the `cdratio` and `cfgwidth` bits of the `ctrl` register in the FPGA manager registers (`fpgamgrregs`) to match the characteristics of the configuration image. These settings are dependent on the MSEL pins input.
2. Set the `nce` bit of the `ctrl` register to 0 to enable HPS configuration.
3. Set the `en` bit of the `ctrl` register to 1 to give the FPGA manager control of the configuration input signals.
4. Set the `nconfigpull` bit of the `ctrl` register to 1 to pull down the `nCONFIG` pin and put the FPGA portion of the device into the reset phase.
5. Poll the `mode` bit of the `stat` register and wait until the FPGA enters the reset phase.
6. Set the `nconfigpull` bit of the `ctrl` register to 0 to release the FPGA from reset.
7. Read the `mode` bit of the `stat` register and wait until the FPGA enters the configuration phase.
8. Clear the interrupt bit of `nSTATUS` (`ns`) in the `gpio_interrupt` register (`fpgamgrregs.mon.gpio_porta_eoi`).
9. Set the `axicfggen` bit of the `ctrl` register to 1 to enable sending configuration data to the FPGA.
10. Write the configuration image to the configuration data register (`data`) in the FPGA manager module configuration data registers (`fpgamgrdata`). You can also choose to use a DMA controller to transfer the configuration image from a peripheral device to the FPGA manager.
11. Use the `fpgamgrregs.mon.gpio_ext_porta` registers to monitor the `CONF_DONE` (`cd`) and `nSTATUS` (`ns`) bits.
12. Set the `axicfggen` bit of the `ctrl` register to 0 to disable configuration data on AXI slave.
13. Clear any previous DONE status by writing a 1 to the `dcntdone` bit of the DCLK status register (`dclkstat`) to clear the completed status flag.
14. Send the DCLKs required by the FPGA to enter the initialization phase.
15. Poll the `dcntdone` bit of the DCLK status register (`dclkstat`) until it changes to 1, which indicates that all the DCLKs have been sent.
16. Write a 1 to the `dcntdone` bit of the DCLK status register to clear the completed status flag.
17. Read the `mode` bit of the `stat` register to wait for the FPGA to enter user mode.
18. Set the `en` bit of the `ctrl` register to 0 to allow the external pins to drive the configuration input signals.

Partial Reconfiguration

Partial reconfiguration allows you to reconfigure part of the device while other sections remain running. The HPS performs partial reconfiguration while the FPGA portion of the device is in user mode. The following sequence suggests one way for software to perform a partial configuration:

- If `PR_READY=1`, continue to step 7.
- If `PR_READY` is 0, then go back and repeat step 5. Note that a minimum of 16 DCLK pulses are required.



If the HPS resets in the middle of a partial reconfiguration, software can assume that the configuration is unsuccessful. After an HPS warm reset, software must repeat the steps for partial configuration. After an HPS cold reset, software must repeat the steps for [Full Configuration](#) on page 29-33.

1. Read the mode bit of the stat register in fpgamgrregs to ensure that the FPGA is in user mode.
2. Set the cdratio bit of the ctrl register to match the characteristics of the partial reconfiguration image and set the cfgwidth bit of the ctrl register to 0 for 16-bit configuration data width.
3. Set the en bit of the ctrl register to 1 to give the FPGA manager control of the configuration input signals.
4. Set the prreq bit of the ctrl register to 1 to assert PR_REQUEST.
5. Write a value of 1 to the dclkcnt register to generate DCLK pulses for one clock cycle.
6. Poll the fpgamgrregs.mon registers to observe the PR_READY (prr) bit.
7. Write a value of 3 to the dclkcnt register to generate DCLK pulses for three clock cycles.
8. Set the axicfggen bit of the ctrl register to 1 to enable sending configuration data to the FPGA.
9. Write the partial reconfiguration image to the data register in the FPGA manager fpgamgrdata registers. You can also choose to use a DMA to transfer the configuration image from a peripheral device to the FPGA manager.
10. Poll the fpgamgrregs.mon registers to observe the PR_DONE (prd), PR_READY (prr), PR_ERROR (pre), and CRC_ERROR (crc) bits until the bits match one of the completion statuses shown in Table A-12.
11. Set the axicfggen bit of the ctrl register to 0 to disable sending configuration data to the FPGA.
12. Set the prreq bit of the ctrl register to 0 to deassert PR_REQUEST.
13. Write a value of 128 to the dclkcnt register to generate DCLK pulses for 128 clock cycles.
14. Poll the dcntdone bit of the dclkstat register until it changes to 1, which indicates that all the DCLKs have been sent.
15. Write a 1 to the dcntdone bit of the dclkstat register to clear the completed status flag.
16. Poll the fpgamgrregs.mon registers to observe the PR_DONE (prd), PR_READY (prr), PR_ERROR (pre), and CRC_ERROR (crc) bits. When all bits are set to 0, the FPGA is ready for the next transaction.
17. Set the en bit of the ctrl register to 0 to allow the external pins to drive the configuration input signals.

Related Information

- [Full Configuration](#) on page 29-33
 - [FPGA Manager](#) on page 5-1
- For more information regarding programming the FPGA through the HPS, refer to the *FPGA Manager* chapter.