Cyclone V Device Handbook

Volume 3: Hard Processor System Technical Reference Manual

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

cv_5v4-1.3

Contents

Chapter 3. Reset Manager

Section II. System Interconnect

Chapter 4. Interconnect

Section IV. Debug and Trace

Chapter 7. CoreSight Debug and Trace

Section V. Memory and Memory Controllers

Chapter 8. SDRAM Controller Subsystem

Chapter 9. On-Chip Memory

Chapter 10. NAND Flash Controller

Chapter 11. SD/MMC Controller

Chapter 12. Quad SPI Flash Controller

Section VI. Peripherals

Chapter 13. FPGA Manager

Chapter 14. System Manager

Chapter 15. Scan Manager

Chapter 16. DMA Controller

Chapter 17. Ethernet Media Access Controller

Chapter 18. USB 2.0 OTG Controller

Chapter 19. SPI Controller

Chapter 20. I²C Controller

Chapter 21. UART Controller

Chapter 22. General-Purpose I/O Interface

Chapter 23. Timer

Chapter 24. Watchdog Timer

Chapter 28. HPS Component Interfaces

Chapter 29. Simulating the HPS Component

Section VIII. Appendices

Appendix A. Booting and Configuration

Additional Information

Chapter Revision Dates

The chapters in this document, Cyclone V Device Handbook, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1. Introduction to the Hard Processor System
Revised: *November 2012*
Part Number: *cv_54001-1.3*

Chapter 2. Clock Manager
Revised: *November 2012*
Part Number: *cv_54002-1.2*

Chapter 3. Reset Manager
Revised: *November 2012*
Part Number: *cv_54003-1.2*

Chapter 4. Interconnect
Revised: *November 2012*
Part Number: *cv_54004-1.2*

Chapter 5. HPS-FPGA AXI Bridges
Revised: *November 2012*
Part Number: *cv_54005-1.1*

Chapter 6. Cortex-A9 Microprocessor Unit Subsystem
Revised: *November 2012*
Part Number: *cv_54006-1.2*

Chapter 7. CoreSight Debug and Trace
Revised: *November 2012*
Part Number: *cv_54007-1.2*

Chapter 8. SDRAM Controller Subsystem
Revised: *November 2012*
Part Number: *cv_54008-1.1*

Chapter 9. On-Chip Memory
Revised: *November 2012*
Part Number: *cv_54009-1.1*

Chapter 10. NAND Flash Controller
Revised: *November 2012*
Part Number: *cv_54010-1.2*

Chapter 11. SD/MMC Controller
Revised: *November 2012*
Part Number: *cv_54011-1.1*

Chapter 12. Quad SPI Flash Controller

Revised: *November 2012*
Part Number: *cv_54012-1.2*

Chapter 13. FPGA Manager
Revised: *November 2012*
Part Number: *cv_54013-1.3*

Chapter 14. System Manager
Revised: *November 2012*
Part Number: *cv_54014-1.2*

Chapter 15. Scan Manager
Revised: *November 2012*
Part Number: *cv_54015-1.2*

Chapter 16. DMA Controller
Revised: *November 2012*
Part Number: *cv_54016-1.1*

Chapter 17. Ethernet Media Access Controller
Revised: *November 2012*
Part Number: *cv_54017-1.2*

Chapter 18. USB 2.0 OTG Controller
Revised: *November 2012*
Part Number: *cv_54018-1.2*

Chapter 19. SPI Controller
Revised: *November 2012*
Part Number: *cv_54019-1.2*

Chapter 20. I^2C Controller
Revised: *November 2012*
Part Number: *cv_54020-1.2*

Chapter 21. UART Controller
Revised: *November 2012*
Part Number: *cv_54021-1.2*

Chapter 22. General-Purpose I/O Interface
Revised: *November 2012*
Part Number: *cv_54022-1.2*

Chapter 23. Timer
Revised: *November 2012*
Part Number: *cv_54023-1.2*

Chapter 24. Watchdog Timer
Revised: *November 2012*
Part Number: *cv_54024-1.2*

Chapter 25. CAN Controller
Revised: *November 2012*
Part Number: *cv_54025-1.2*

Chapter 26. Introduction to the HPS Component
Revised: *June 2012*
Part Number: *cv_54026-1.0*

Chapter 27. Instantiating the HPS Component
Revised: *November 2012*
Part Number: *cv_54027-1.1*

Chapter 28. HPS Component Interfaces
Revised: *November 2012*
Part Number: *cv_54028-1.1*

Chapter 29. Simulating the HPS Component
Revised: *November 2012*
Part Number: *cv_54030-1.1*

Appendix A. Booting and Configuration
Revised: *November 2012*
Part Number: *cv_5400A-1.3*

Section I. Overview

This section includes the following chapters:

- Chapter 1, Introduction to the Hard Processor System
- Chapter 2, Clock Manager
- Chapter 3, Reset Manager

For information about the revision history for chapters in this section, refer to "Document Revision History" in each individual chapter.

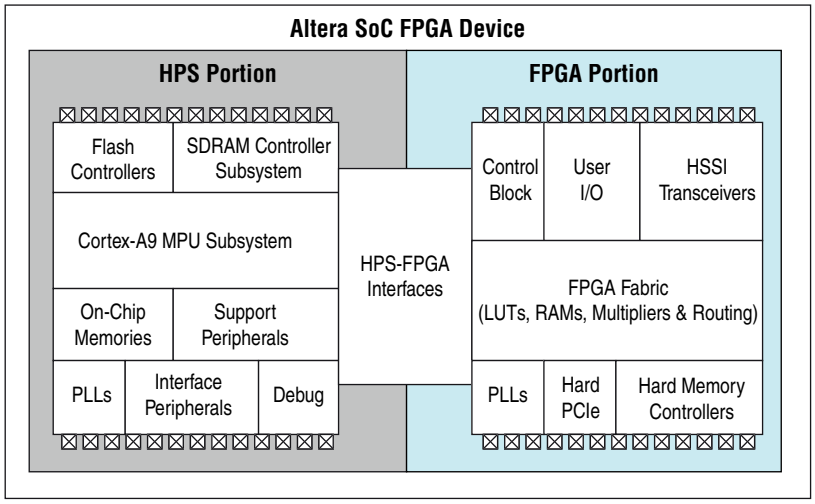
1. Introduction to the Hard Processor System

cv_54001-1.3

The Cyclone® V SoC FPGA device is a single-die system on a chip (SoC) that consists of two distinct parts—a hard processor system (HPS) portion and an FPGA portion.

Figure 1–1 shows a high-level block diagram of the Altera SoC FPGA device. Blocks connected to device pins have symbols (square with X) adjacent to them in the figure.

Figure 1–1. Altera SoC FPGA Device Block Diagram



The HPS contains a microprocessor unit (MPU) subsystem with single or dual ARM® Cortex™-A9 MPCore processors, flash memory controllers, an SDRAM controller subsystem, on-chip memories, support peripherals, interface peripherals, debug capabilities, and PLLs. The dual-processor HPS supports symmetric (SMP) and asymmetric (AMP) multiprocessing.

The FPGA portion of the device contains the FPGA fabric, a control block (CB), phase-locked loops (PLLs), and depending on the device variant, high-speed serial interface (HSSI) transceivers, hard PCI Express® (PCIe®) controllers, and hard memory controllers.

For information about the FPGA portion of the device, refer to *Cyclone V Device Overview*.

The HPS and FPGA portions of the device are distinctly different. The HPS boots (from any of multiple boot sources, including the FPGA fabric and external flash devices) and the FPGA gets configured (through the HPS or any external source supported by the device).

For more information, refer to the *Booting and Configuration* appendix in volume 3 of the *Cyclone V Device Handbook*.

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 software executing 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. Or, when the FPGA portion is configured by an external source, the MPU subsystem can boot from memory 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. But to power on the FPGA portion, the HPS must already be on or powered on 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.

Features of the HPS

The following list contains the main modules of the HPS:

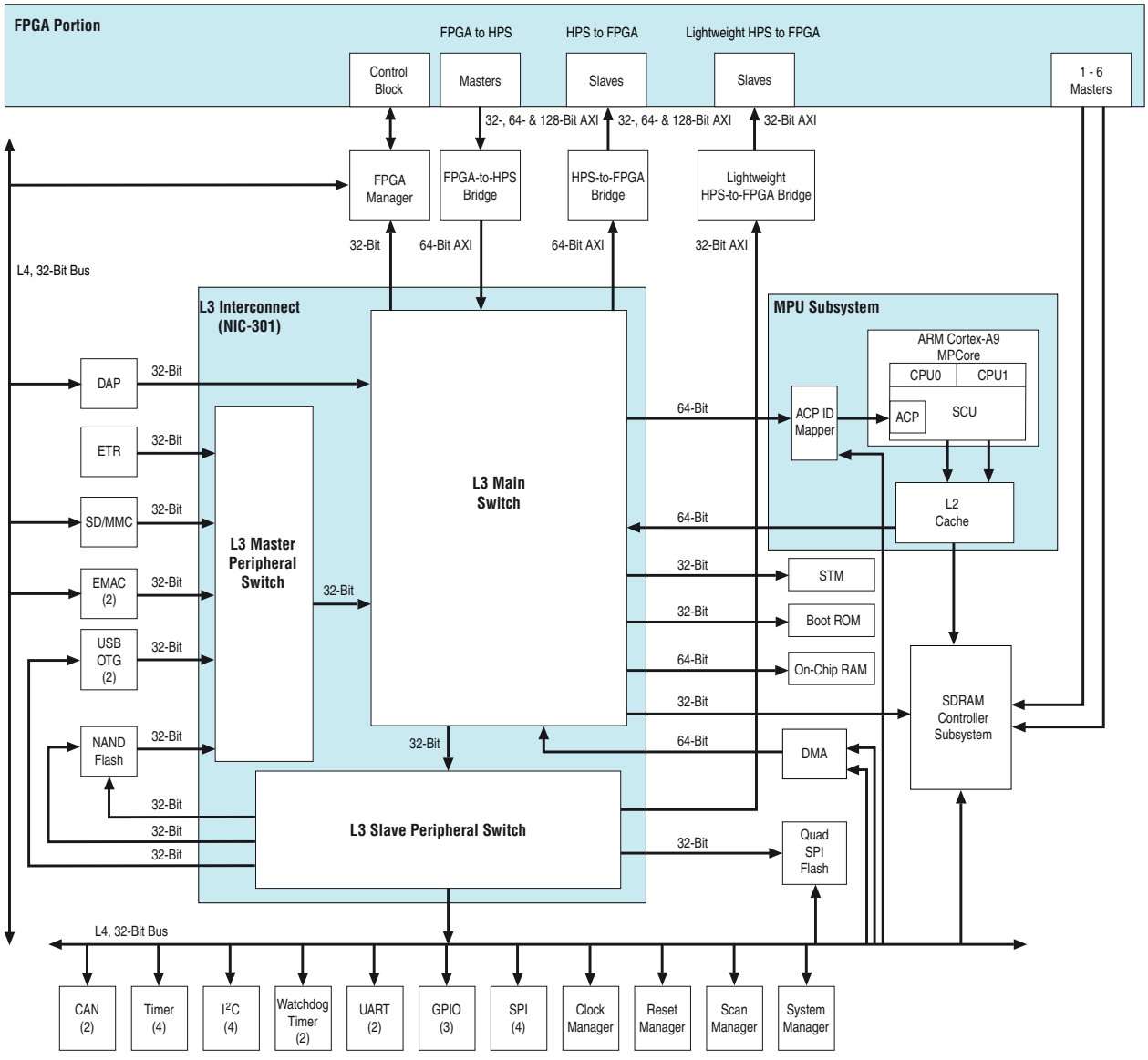
- An MPU subsystem featuring dual ARM Cortex-A9 MPCore processors
- An SDRAM controller subsystem
- One general-purpose direct memory access (DMA) controller
- Two Ethernet media access controllers (EMACs)
- Two USB 2.0 On-The-Go (OTG) controllers
- One NAND flash controller
- One quad SPI flash controller
- One Secure Digital (SD) / MultiMediaCard (MMC) controller
- Two serial peripheral interface (SPI) master controllers
- Two SPI slave controllers
- Four inter-integrated circuit (I^2C) controllers
- 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
- Two controller area network (CAN) controllers (certain device variants only)
- 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)
- A system manager
- A clock manager
- A reset manager
- A scan manager
- An FPGA manager
- One FPGA-to-HPS bridge
- Two HPS-to-FPGA bridges

HPS Block Diagram and System Integration

Figure 1–2 shows a block diagram of most modules in the HPS. Debug modules are not shown.

Figure 1–2. HPS Block Diagram



The following sections list features of the HPS modules and reference related chapters that provide more detailed information. The HPS incorporates third-party intellectual property (IP) from several vendors. Each chapter in this handbook identifies additional third-party IP documentation, if available from the third-party vendors.

No clock information is listed in the following summary sections. For comprehensive clock information for all modules, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

MPU Subsystem

The MPU subsystem provides the following functionality:

- ARM Cortex-A9 MPCore
  - One or two ARM Cortex-A9 processors in a cluster
  - NEON™ SIMD coprocessor and VFPv3 per processor
  - Snoop Control Unit (SCU) to ensure coherency within the cluster
  - 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

As shown in Figure 1–2, the L2 cache has one 64-bit master port connected to the L3 interconnect and one 64-bit master port connected directly to the SDRAM controller subsystem. A programmable address filter in the L2 cache controls which portions of the 32-bit physical address space use which master.

For more information, refer to the *Cortex-A9 MPU System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Interconnect

The interconnect consists of the L3 interconnect and level 4 (L4) buses. The L3 interconnect is one 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

The L4 buses are each connected to a master in the L3 slave peripheral switch. Each L4 bus is 32 bits wide and is connected to multiple slaves. Each L4 bus operates on a separate clock source.

For more information, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

Memory Controllers

The HPS provides the memory controllers described in this section.

SDRAM Controller Subsystem

The SDRAM controller subsystem is mastered by HPS masters and FPGA fabric masters. The FPGA-to-HPS SDRAM interface is compatible with the hard memory controllers in the FPGA portion of the device and with hard SDRAM controllers in non-HPS FPGA devices, such as Stratix IV FPGAs.

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
- Software-configurable priority scheduling on individual SDRAM bursts
- 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
- Support for ARM Advanced Microcontroller Bus Architecture (AMBA®) Advanced eXtensible Interface (AXI™) quality of service (QoS) for the fabric interfaces

The SDRAM controller subsystem is composed of the SDRAM controller and the DDR PHY.

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 via 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

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

- Command reordering (look-ahead bank management)
- Data reordering (out of order transactions)
- Deficit round-robin arbitration with aging for bandwidth management
- High-priority bypass for latency sensitive traffic

DDR PHY

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

For more information, refer to the *SDRAM Controller Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*.

NAND Flash Controller

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

- Supports single-level cell (SLC) and multilevel cell (MLC) NAND flash devices
- Integrated descriptor-based DMA controller
- 8-bit ONFI 1.0 NAND flash devices
- Programmable page sizes of 512 bytes, 2 KB, 4 KB, and 8 KB
- Supports 32, 64, 128, 256, 384, and 512 pages per block
- Programmable hardware ECC for SLC and MLC devices
  - 512 bytes ECC sector size with 4-, 8-, or 16-bit correction
  - 1 KB ECC sector size with 24-bit correction

For more information, refer to the *NAND Flash Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

Quad SPI Flash Controller

The quad SPI flash controller is based on Cadence Quad SPI Flash Controller (QSPI_FLASH_CTRL) 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 I/O, dual I/O, quad I/O instructions
- Programmable data frame size of 8, 16, or 32 bits
- Support up to four chip selects

For more information, refer to the *Quad SPI Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

SD/MMC Controller

The SD/MMC controller is based on Synopsys® DesignWare® Mobile Storage Host (DWC_mobile_storage) 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: x1, x4, or x8
  - Programmable card types: SD, SDIO, or MMC version 4.3 and 4.4 devices
- Up to 64 KB programmable block size

For more information, refer to the *SD/MMC Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

Support Peripherals

HPS provides the support peripherals described in this section.

Clock Manager

The clock manager offers the following features:

- Manages clocks for HPS
- Supports dynamic clock tuning

For more information, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Reset Manager

The reset manager offers the following features:

- Manages reset for HPS

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

System Manager

The system manager offers the following features:

- ECC monitoring and control
- Pin multiplexing
- 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

For more information, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Scan Manager

The scan manager offers the following features:

- Drives serial scan-chains to FPGA JTAG and HPS I/O bank configuration

For more information, refer to the *Scan Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Timers

The four timers are based on the Synopsys DesignWare APB Timers (DW_apb_timers) peripheral and offer the following features:

- 32-bit timer resolution
- Support free-running timer mode
- Programmable time-out period up to approximately 86 seconds (assuming a 50 MHz clock)
- Interrupt generation

For more information, refer to the *Timer* chapter in volume 3 of the *Cyclone V Device Handbook*.

Watchdog Timers

The two watchdog timers are based on the Synopsys DesignWare APB Watchdog Timer (DW_apb_wdt) 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 clock)

For more information, refer to the *Watchdog Timer* chapter in volume 3 of the *Cyclone V Device Handbook*.

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:

- Microcoded to support flexible transfer types
- Supports up to 8 channels
- Supports flow control with 31 peripherals handshake interfaces

For more information, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

FPGA Manager

The FPGA manager offers the following features:

- Manages configuration of the FPGA portion of the device
- Mimics passive parallel 32-bit configuration
- 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

For more information, refer to the *FPGA Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Interface Peripherals

HPS provides the interface peripherals described in this section.

EMACs

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

- Supports 10-, 100-, and 1000-Mbps standards
- Supports RGMII external PHY interface
- Integrated DMA controllers

For more information, refer to the *Ethernet Media Access Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

USB Controllers

The two USB 2.0 On-The-Go (OTG) controllers are based on the Synopsys DesignWare Cores USB 2.0 Hi-Speed On-The-Go (DWC_otg) controller and offer the following features:

- Supports USB 2.0 host and device operation
- Dual-role device (device and host functions)
  - High-speed (480 Mbps)
  - Full-speed (12 Mbps)
  - Low-speed (1.5 Mbps)
  - Supports USB 1.1 (full-speed & low-speed)
- Integrated descriptor-based scatter-gather DMA (SGDMA)
- Support for external ULPI PHY
- Up to 16 bidirectional endpoints, including control endpoint

- Up to 16 host channels
- Supports generic root hub
- Automatic ping capability
- Configurable to OTG 1.3 and OTG 2.0 modes

For more information, refer to the *USB 2.0 OTG Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

I²C Controllers

The four I²C controllers are based on Synopsys DesignWare APB I²C (DW_apb_i2c) controller and offer the following features:

- Two controllers support I²C management interfaces
- Support both 100 KBps and 400 KBps modes
- Support both 7-bit and 10-bit addressing modes
  - No support for mixed-address mode
- Support master and slave operating mode
- Direct access for host processor
  - DMA controller may be used for large transfers

For more information, refer to the *I²C Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

UARTs

The two UART modules are based on Synopsys DesignWare APB Universal Asynchronous Receiver/Transmitter (DW_apb_uart) peripheral and offer the following features:

- 16550 compatible UART
  - Supports the auto flow control as specified in 16750 specification
- Supports IrDA 1.0 SIR mode
- Programmable baud rate up to 115.2 Kbps
- Direct access for host processor
  - DMA controller may be used for large transfers

For more information, refer to the *UART Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

CAN Controllers

The two CAN controllers are based on the Bosch® D_CAN controller and offer the following features:

- Compliant with CAN protocol specification 2.0 part A & B
- Programmable communication rate up to 1 Mbps

- Holds up to 128 messages
- Supports 11-bit standard and 29-bit extended identifiers
- Programmable interrupt scheme
- Direct access for host processor
  - DMA controller may be used for large transfers
- Available on certain device variants only

For more information, refer to the *Controller Area Network Controller* chapter in volume 4 of the *Cyclone V Device Handbook*.

SPI Master Controllers

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

- Programmable data frame size from 4 to 16 bits
- Supports full and half duplex
- Support up to two chip selects
- Direct access for host processor
  - DMA controller may be used for large transfers

For more information, refer to the *SPI Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

SPI Slave Controllers

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

- Programmable data frame size from 4 to 16 bits
- Supports full and half duplex
- Direct access for host processor
  - DMA controller may be used for large transfers

For more information, refer to the *SPI Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

GPIO Interfaces

The three GPIO interfaces are based on Synopsys DesignWare APB General Purpose Programming I/O (DW_apb_gpio) 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

For more information, refer to the *General-Purpose I/O Interface* chapter in volume 3 of the *Cyclone V Device Handbook*.

On-Chip Memory

On-chip memory consists of the two modules described in this section.

On-Chip RAM

The on-chip RAM offers the following features:

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

For more information, refer to the *On-Chip Memory* chapter in volume 3 of the *Cyclone V Device Handbook*.

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

For more information, refer to the *On-Chip Memory* chapter in volume 3 of the *Cyclone V Device Handbook*.

Endian Support

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

The processors masters are software configurable to interpret data as little-endian or big-endian, byte-invariant (BE8). All other masters, including the USB interface, are little-endian.

The FPGA-to-HPS, HPS-to-FPGA, 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 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.

HPS-FPGA Interfaces

The HPS-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. The HPS-FPGA interfaces include:

- FPGA-to-HPS bridge—a high-performance AXI bus with a configurable data width of 32-, 64-, and 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 interface.

  For information about the coherent memory interface, refer to the *Cortex-A9 MPU System* chapter in volume 3 of the *Cyclone V Device Handbook*.

- HPS-to-FPGA bridge—a high-performance AXI bus with a configurable data width of 32-, 64-, and 128-bits, allowing the HPS to master transactions to slaves in the FPGA fabric.
- Lightweight HPS-to-FPGA bridge—an AXI bus with a 32-bit fixed data width, allowing the HPS to master transactions to slaves in the FPGA fabric.
- FPGA-to-HPS SDRAM interface—a configurable interface to the MPFE of the SDRAM controller. You can configure the following parameters:
  - AXI-3 or Avalon® Memory-Mapped (Avalon-MM) protocol
  - Up to six ports
  - 32-, 64-, 128-, or 256-bit data width of each port
- FPGA clocks and resets—provide flexible clocks to and from the HPS.
- HPS-to-FPGA JTAG—allows the HPS to master the FPGA JTAG chain.
- TPIU trace—sends trace data created in the HPS to the FPGA fabric.
- FPGA System Trace Macrocell (STM) events—an interface that allows the FPGA fabric to send hardware events stored in the HPS trace using STM.
- FPGA cross-trigger—an interface that allows triggers to and from the CoreSight trigger system.
- DMA peripheral interface—multiple peripheral-request channels.
- FPGA manager interface—signals that communicate with FPGA fabric for boot and configuration.
- Interrupts—allow soft IP 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.

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.

Address Spaces

Table 1–1 shows the HPS address spaces and their sizes.

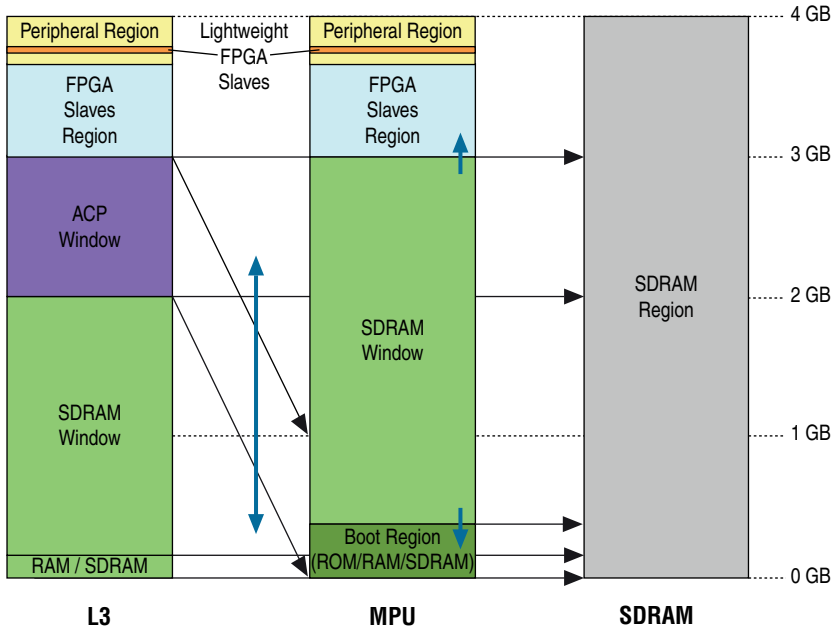
Table 1–1. HPS Address Spaces

Name	Description	Size
MPU	MPU subsystem	4 GB
L3	L3 interconnect	4 GB
SDRAM	SDRAM controller subsystem	4 GB

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

Figure 1–3 shows the relationships between the HPS address spaces. The figure is not to scale.

Figure 1–3. 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.

Table 1–2 shows the base address and size of each region that is common to the L3 and MPU address spaces.

Table 1–2. Common Address Space Regions

Identifier	Region Name	Base Address	Size
FPGASLAVES	FPGA slaves	0xC0000000	960 MB
LWFPGASLAVES	Lightweight FPGA slaves	0xFF200000	2 MB
PERIPH	Peripheral	0xFC000000	64 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 varies. For specific details, refer to "MPU Address Space" and "L3 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 MPU L2 cache controller contains a master connected to the L3 interconnect and a master connected to the SDRAM. 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 L3 interconnect master.

  Figure 1–3 shows the reset values of the SDRAM window boundaries. By default, processor accesses to locations between 0x100000 (1 MB) to 0xC0000000 (3 GB) are made to the SDRAM controller, accesses to all other locations are made to the L3 interconnect. 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.

  For more information about L2 address filtering, refer to the *Cortex-A9 MPU System* chapter in volume 3 of the *Cyclone V Device Handbook*.

- The boot region is 1 MB starting at address 0x0 and is visible to the MPU only when the L2 address filter start register is set to 0x100000. The L3 interconnect Global Programmers View (GPV) remap control register determines if the boot region is mapped to the on-chip RAM or the boot ROM.

  For information about the L3 GPV remap control register bits, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

  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.

- The FPGA slaves region provides access to 960 MB of slaves in the FPGA fabric through the HPS-to-FPGA bridge. If the top of the SDRAM window increases in the MPU address space (by writing to the L2 address filter end register), the lower portion of the FPGA slaves region is obscured from the MPU subsystem.
- The peripheral region contains 64 MB at the top of the address space. The peripheral region includes all slaves connected to the L3 interconnect, L4 buses, and internally-decoded MPU registers (SCU and L2). The boot ROM and on-chip RAM are always mapped into the peripheral region (independent of the boot region contents). The lightweight FPGA slaves are also mapped in the peripheral region and provide access to 2 MB of slaves in the FPGA fabric through the lightweight HPS-to-FPGA bridge.

Table 1–3 shows the base address and size of each MPU address space region that is not included in Table 1–2.

Table 1–3. MPU Default Address Space Regions

Identifier	Region Name	Base Address	Size
MPUBOOT	Boot region	0x00000000	1 MB
MPUSDRAM	SDRAM window	0x00100000	3071 MB

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 following regions:

- The peripheral region is the same as the peripheral region in the MPU address space except that the boot ROM and internal MPU registers (SCU and L2) are not accessible.
- The FPGA slaves region provides access to 960 MB of slaves in the FPGA fabric through the HPS-to-FPGA bridge.
- The SDRAM window region is 2 GB and provides access to the bottom 2 GB of the SDRAM address space. The L3 interconnect GPV `remap` register 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.

  For information about the L3 GPV remap control register bits, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

- 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 MPU System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Table 1–4 shows the base address and size of each L3 address space region that is not included in Table 1–2.

Table 1–4. L3 Address Space Regions

Identifier	Region Name	Base Address	Size
L3SDRAM	SDRAM window	0x00000000	2 GB
L3LOWOCRAM	On-chip RAM when present	0x00000000	64 KB
L3ACP	ACP window	0x80000000	1 GB

Peripheral Region Address Map

Table 1–5 lists the slave identifier, slave title, base address, and size of each slave in the peripheral region. The Slave Identifier column lists the names used in the HPS register map. The Slave Title column contains the module name for modules with only one slave and module names plus a suffix for modules with more than one slave.

Table 1–5. Peripheral Region Address Map (Part 1 of 2)

Slave Identifier	Slave Title	Base Address	Size
STM	STM	0xFC000000	48 MB
DAP	DAP	0xFF000000	2 MB
LWFPGASLAVES	FPGA slaves accessed with lightweight FPGA-to-HPS AXI bridge	0xFF200000	2 MB
LWHPS2FPGAREGS	Lightweight FPGA-to-HPS AXI bridge GPV	0xFF400000	1 MB
HPS2FPGAREGS	HPS-to-FPGA AXI bridge GPV	0xFF500000	1 MB
FPGA2HPSREGS	FPGA-to-HPS AXI bridge GPV	0xFF600000	1 MB
EMAC0	EMAC0	0xFF700000	8 KB
EMAC1	EMAC1	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	0xFF207000	4 KB
GPIO0	GPIO0	0xFF208000	4 KB
GPIO1	GPIO1	0xFF209000	4 KB
GPIO2	GPIO2	0xFF20A000	4 KB
L3REGS	L3 interconnect GPV	0xFF800000	1 MB
NANDDATA	NAND controller data	0xFF900000	1 MB
QSPIDATA	Quad SPI flash data	0xFFA00000	1 MB
USB0	USB0 OTG controller registers	0xFFB00000	256 KB
USB1	USB1 OTG controller registers	0xFFB40000	256 KB
NANDREGS	NAND controller registers	0xFFB80000	64 KB
FPGAMGRDATA	FPGA manager configuration data	0xFFB90000	4 KB
CAN0	CAN0 controller registers	0xFFC00000	4 KB
CAN1	CAN1 controller registers	0xFFC01000	4 KB

Table 1–5. Peripheral Region Address Map (Part 2 of 2)

Slave Identifier	Slave Title	Base Address	Size
UART0	UART0	0xFFC02000	4 KB
UART1	UART1	0xFFC03000	4 KB
I2C0	I2C0	0xFFC04000	4 KB
I2C1	I2C1	0xFFC05000	4 KB
I2C2	I2C2	0xFFC06000	4 KB
I2C3	I2C3	0xFFC07000	4 KB
SPTIMER0	SP Timer0	0xFFC08000	4 KB
SPTIMER1	SP Timer1	0xFFC09000	4 KB
SDRREGS	SDRAM controller subsystem registers	0xFFC20000	128 KB
OSC1TIMER0	OSC1 Timer0	0xFFD00000	4 KB
OSC1TIMER1	OSC1 Timer1	0xFFD01000	4 KB
L4WD0	Watchdog0	0xFFD02000	4 KB
L4WD1	Watchdog1	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 slave0	0xFFE02000	4 KB
SPIS1	SPI slave1	0xFFE03000	4 KB
SPIM0	SPI master0	0xFFF00000	4 KB
SPIM1	SPI master1	0xFFF01000	4 KB
SCANMGR	Scan manager registers	0xFFF02000	4 KB
ROM	Boot ROM	0xFFFD0000	64 KB
MPUSCU	MPU SCU registers	0xFFFEC000	8 KB
MPUL2	MPU L2 cache controller registers	0xFFFEF000	4 KB
OCRAM	On-chip RAM	0xFFFF0000	64 KB

Document Revision History

Table 1–6 shows the revision history for this document.

Table 1–6. Document Revision History

Date	Version	Changes
November 2012	1.3	Minor updates.
June 2012	1.2	Updated address spaces section.
May 2012	1.1	Added peripheral region address map.
January 2012	1.0	Initial release.

2. Clock Manager

cv_54002-1.2

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:
  - Main—contains clocks for the Cortex™-A9 microprocessor unit (MPU) subsystem, level 3 (L3) interconnect, level 4 (L4) peripheral bus, and debug
  - 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
- 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

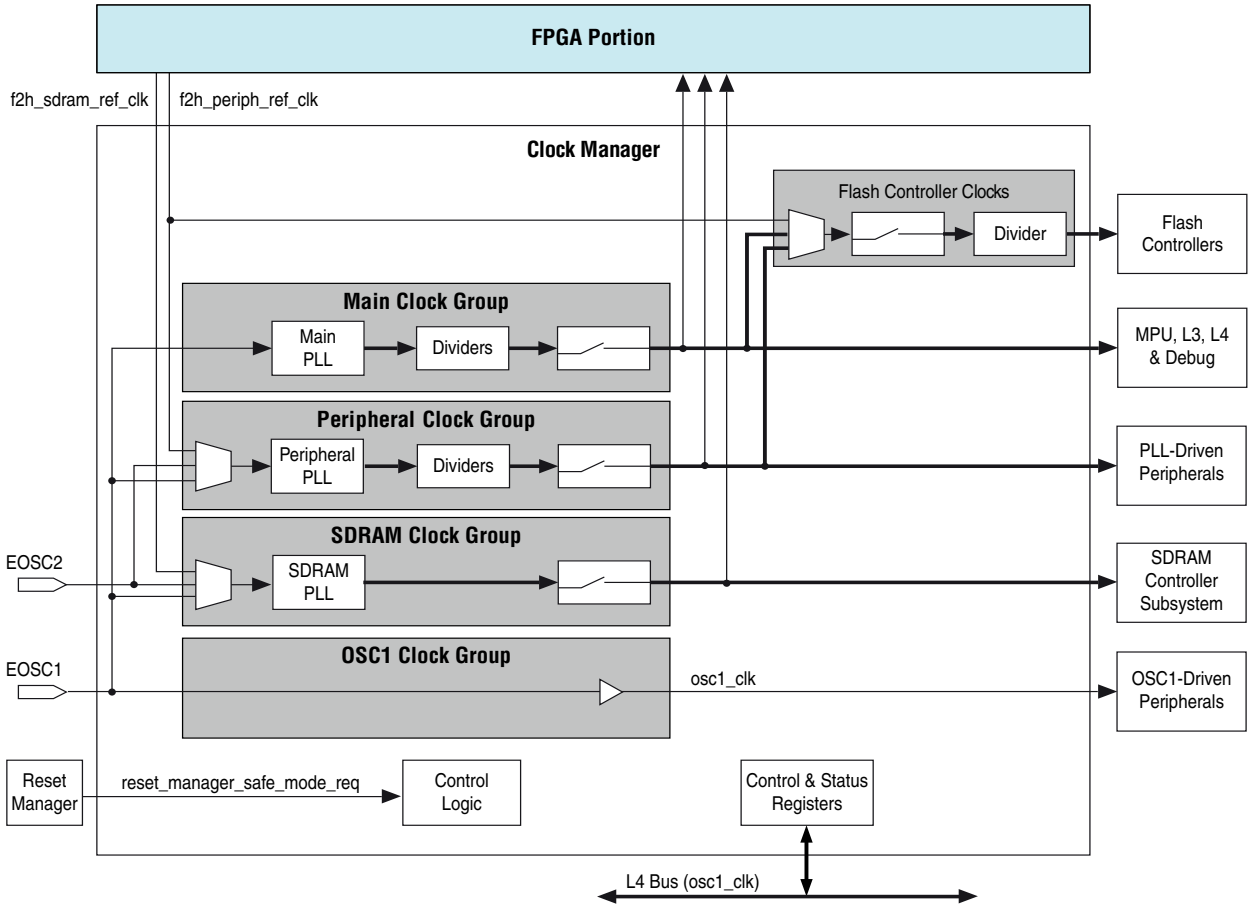
The clock manager is ***not*** responsible for the following functional behaviors:

- Selection or management of the clocks for the FPGA-to-HPS, HPS-to-FPGA, and FPGA-to-HPS SDRAM 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" on page 2–5.

Clock Manager Block Diagram and System Integration

Figure 2–1 shows the major components of the clock manager and its integration in the HPS.

Figure 2–1. Clock Manager Block Diagram



The next section describes the functional blocks internal to the clock manager. Refer to Figure 2–3 to Figure 2–6 for more detailed versions of the grey boxes in Figure 2–1.

Functional Description of the Clock Manager

This section describes the functional operation of the clock manager.

Clock Manager Building Blocks

The clock manager has the following major building blocks.

PLLs

The clock manager contains three PLLs: main, 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
  - Multiplier range is 1 to 4096
  - 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

Equation 2–1 shows equations for F_{REF}, F_{VCO}, and F_{OUT}. The values for *M*, *N*, and *C* are stored in registers accessible to software.

Equation 2–1. F_{REF}, F_{VCO}, and F_{OUT} Equations

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

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

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

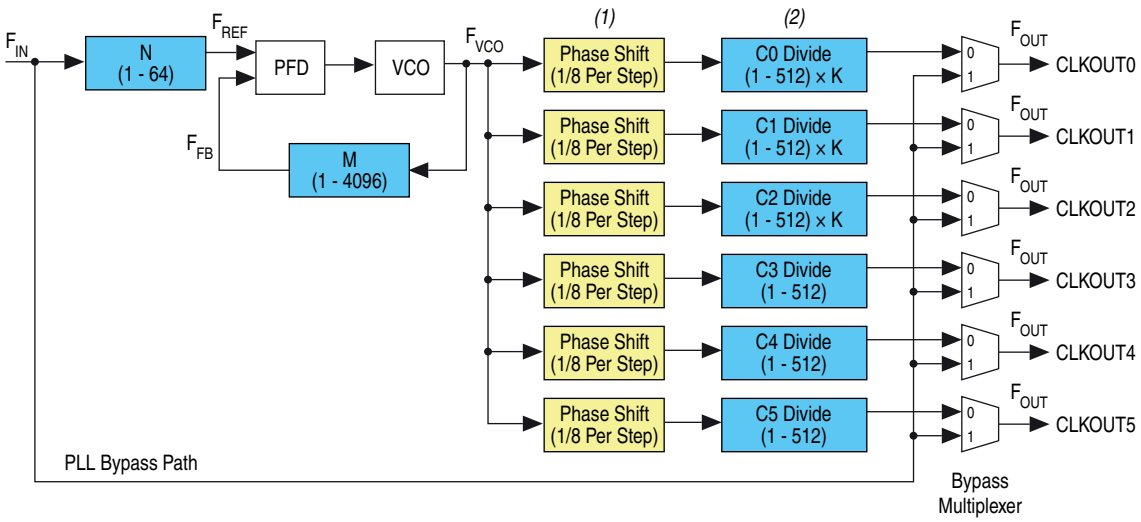
where:

(1) F_{VCO} = VCO frequency.
(2) F_{IN} = input frequency.
(3) F_{REF} = reference frequency.
(4) M = numerator, part of the clock feedback path.
(5) N = denominator, part of the input clock reference path.
(6) C_i = post-scale counter, where *i* is 0-5 for each of the six counters.
(7) K is an internal post-scale counter in the main PLL, where K = 2 for C0, and K = 4 for C1 and C2. K = 1 for C3, C4, and C5 in the main PLL and for all C_i counters in the peripheral and SDRAM PLLs.

Minimum and maximum VCO frequencies for the main, peripheral, and SDRAM PLLs vary by device speed grade. For specific details, refer the *Cyclone V Device Datasheet*.

Figure 2–2 shows the block diagram of each PLL. Values listed for *M*, *N*, and *C* are actually one greater than the values stored in the CSRs.

Figure 2–2. PLL Block Diagram



Notes to Figure 2–2:

(1) Phase shift is only available for SDRAM PLL outputs.

(2) In the main PLL, K=2 for C0, and K=4 for C1 and C2. K=1 in the peripheral and SDRAM PLLs.

Dividers

Dividers subdivide the C0-C5 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.

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 off any clock affected by the change, wait for any PLL lock if required, then gate the clocks back on. 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 EOSC1 pin.

OSC1 Clock Group

The clock in the OSC1 clock group is derived directly from the EOSC1 pin. This clock is never gated or divided. It is used as a PLL input and also by HPS logic that does not operate on a clock output from a PLL.

Table 2–1 lists the clock in the OSC1 clock group.

Table 2–1. OSC1 Clock Group Clock

Name	Frequency	Clock Source	Destination
osc1_clk	10 to 50 MHz	EOSC1 pin	OSC1-driven peripherals listed in Table 2–9 on page 2–14

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 EOSC1 pin of the device.

Table 2–2 lists the main PLL output assignments.

Table 2–2. Main PLL Output Assignments

PLL	Output Counter	Clock Name	Frequency	Phase Shift Control
Main	C0	mpu_base_clk	osc1_clk to varies (1)	No
	C1	main_base_clk	osc1_clk to varies (1)	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

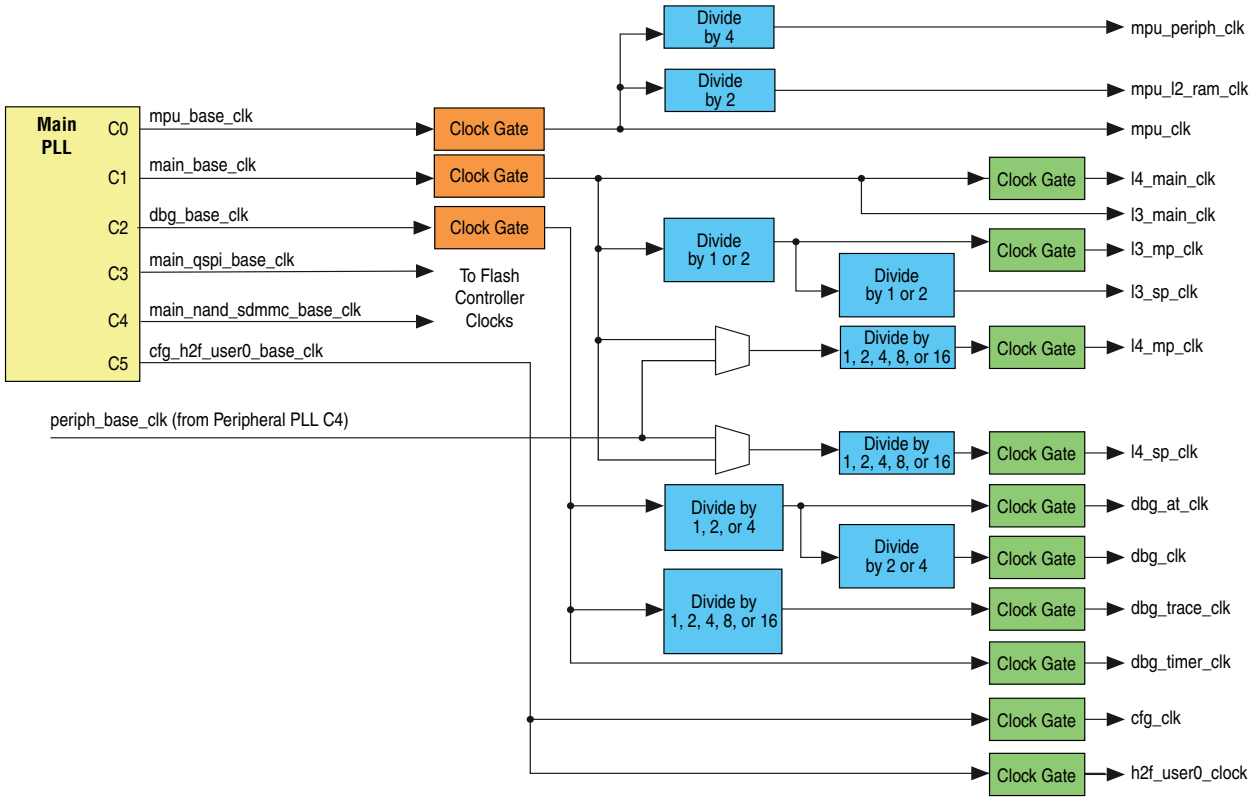
Note to Table 2–2:

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

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, 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.

Figure 2–3 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.

Figure 2–3. 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 2–3 lists the clocks in the main clock group.

Table 2–3. 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
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 or 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	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 125_MHz divided from main PLL C5	FPGA manager configuration clock
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 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 `EOSC1` pin, the `EOSC2` pin, or the `f2h_periph_ref_clk` clock provided by the FPGA fabric.

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, 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 2–4 lists the Peripheral PLL output assignments.

Table 2–4. 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

Figure 2–4 shows programmable post-PLL dividers and clock gating for the peripheral 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 2–4. Peripheral Clock Group Divide and Gating

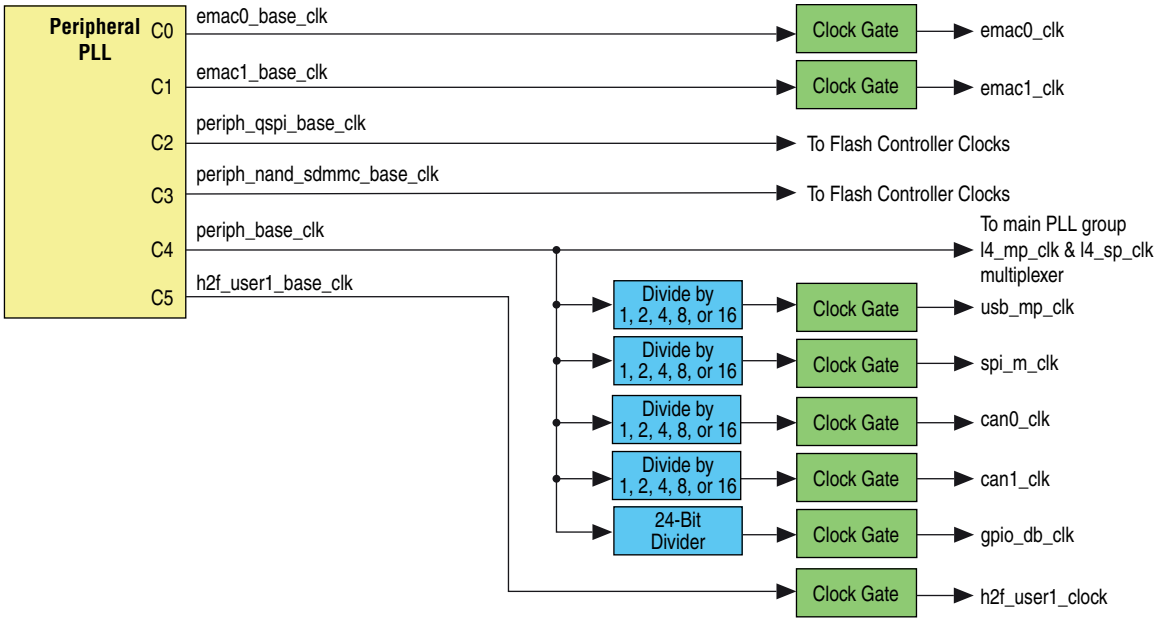


Table 2–5 lists the clocks in the peripheral clock group.

Table 2–5. Peripheral Clock Group Clocks (Part 1 of 2)

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.
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
can0_clk	Up to 100 MHz	Peripheral PLL C4	Controller area network (CAN) controller 0 clock

Table 2–5. Peripheral Clock Group Clocks (Part 2 of 2)

System Clock Name	Frequency	Divided From	Constraints and Notes
can1_clk	Up to 100 MHz	Peripheral PLL C4	CAN controller 1 clock
gpio_db_clk	Up to 32 KHz	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

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 EOSC1 pin, the EOSC2 pin, or the f2h_sdram_ref_clk clock provided by the FPGA fabric.

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 2–6 lists the SDRAM PLL output assignments.

Table 2–6. SDRAM PLL Output Assignments

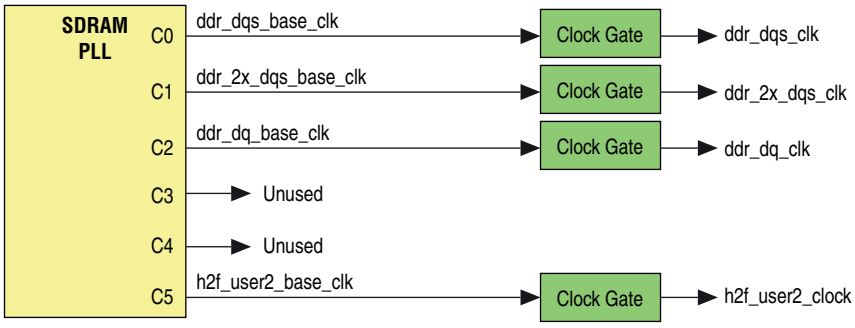
PLL	Output Counter	Clock Name	Frequency	Phase Shift Control
SDRAM	C0	ddr_dqs_base_clk	Up to varies (1)	Yes
	C1	ddr_2x_dqs_base_clk	Up to ddr_dqs_base_clk x 2	Yes
	C2	ddr_dq_base_clk	Up to ddr_dqs_base_clk	Yes
	C5	h2f_user2_base_clk	osc1_clk to varies (1)	Yes

Note to Table 2–6:

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

Figure 2–5 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 2–5. 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 2–7 lists the clocks in the SDRAM clock group.

Table 2–7. 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
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

Flash Controller Clocks

Flash memory peripherals can be driven by the main PLL, the peripheral PLL, or from clocks provided by the FPGA fabric, as shown in Figure 2–6.

Figure 2–6. Flash Peripheral Clock Divide and Gating

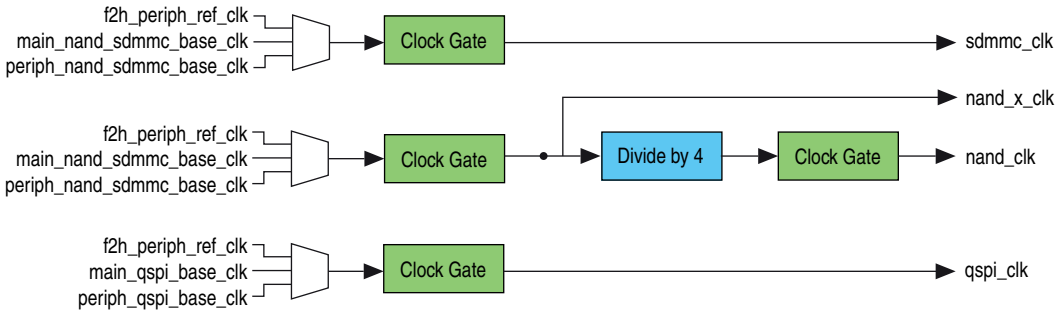


Table 2–8 lists the flash controller clocks.

Table 2–8. 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
sdmmc_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 26 and 52 MHz ■ SD/MMC has a subclock tree divided down from this clock

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.

For more information, refer to "Safe Mode".

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 on all modules that receive warm reset.

For more information, refer to "Reset Sequencing" in the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Safe Mode

Safe mode is enabled in the HPS by the assertion of a safe mode request from the reset manager or by a cold reset. Assertion of the safe mode request from the reset manager sets the safe mode bit in the clock manager control register. No other control register bits are affected by the safe mode request from the reset manager.

When safe mode is enabled, the main PLL hardware-managed clocks (C0-C2) are bypassed to the `osc1_clk` clock and are directly generated from the `osc1_clk` clock. 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 the `osc1_clk` clock, 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.

Before coming out of safe mode, all registers and clocks need to 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. Altera strongly recommends using Altera-provided libraries to configure and control HPS clocks.

Interrupts

The clock manger provides one interrupt output, enabled using the interrupt enable register (`intren`). The source of the interrupt is six inputs, namely, an achieving lock and a losing lock bit in the interrupt status register (`inter`) for each PLL.

Clock Usage By Module

Table 2–9 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 2–9. Clock Usage By Module (Part 1 of 3)

Module Name	System Clock Name	Use
MPU subsystem	mpu_clk	Main clock for the MPU subsystem
	mpu_periph_clk	Clock for peripherals inside the MPU subsystem
	dbg_at_clk	Trace bus clock
	dbg_clk	Debug clock
	mpu_l2_ram_clk	Clock for the L2 cache and Accelerator Coherency Port (ACP) ID mapper
	l4_mp_clk	Clock for the ACP ID mapper control slave
Interconnect	l3_main_clk	Clock for the L3 main switch
	dbg_at_clk	Clock for the System Trace Macrocell (STM) slave and Embedded Trace Router (ETR) master connections
	dbg_clk	Clock for the DAP master connection
	l3_mp_clk	Clock for the L3 master peripheral switch
	l4_mp_clk	Clock for the L4 MP bus, Secure Digital (SD) / MultiMediaCard (MMC) master, and EMAC masters
	usb_mp_clk	Clock for the USB masters and slaves
	nand_x_clk	Clock for the NAND master
	cfg_clk	Clock for the FPGA manager configuration data slave
	l3_sp_clk	Clock for the L3 slave peripheral switch
	l3_main_clk	Clock for the L4 SPIS bus master
	mpu_l2_ram_clk	Clock for the ACP ID mapper slave and L2 master connections
	osc1_clk	Clock for the L4 OSC1 bus master
	spi_m_clk	Clock for the L4 SPIM bus master
	l4_sp_clk	Clock for the L4 SP bus master
	l4_mp_clk	Clock for the quad SPI bus slave
Boot ROM	l3_main_clk	Clock for the boot ROM
On-chip RAM	l3_main_clk	Clock for the on-chip RAM

Table 2–9. Clock Usage By Module (Part 2 of 3)

Module Name	System Clock Name	Use
DMA controller	l4_main_clk	Clock for the DMA
	dbg_at_clk	Clock synchronous to the STM module
	l4_mp_clk	Clock synchronous to the quad SPI flash
FPGA manager	cfg_clk	Clock for the control block (CB) data interface and configuration data slave
	l4_mp_clk	Clock for the control slave
HPS-to-FPGA bridge	l3_main_clk	Clock for the data slave
	l4_mp_clk	Clock for the global programmer's view (GPV) slave
FPGA-to-HPS bridge	l3_main_clk	Clock for the data master
	l4_mp_clk	Clock for the GPV slave
Lightweight HPS-to-FPGA bridge	l4_mp_clk	Clock for the GPV masters, and the data and GPV slave
Quad SPI flash controller	l4_mp_clk	Clock for the control slave
	qspi_clk	Reference clock for serialization
SD/MMC controller	l4_mp_clk	Clock for the master and slave
	sdmmc_clk	Clock for the SD/MMC internal logic
EMAC 0	l4_mp_clk	Clock for the master
	emac0_clk	EMAC 0 internal logic clock
	osc1_clk	IEEE 1588 timestamp clock
EMAC 1	l4_mp_clk	Clock for the master
	emac1_clk	EMAC 1 internal logic clock
	osc1_clk	IEEE 1588 timestamp clock
USB 0	usb_mp_clk	Clock for the master and slave
USB 1	usb_mp_clk	Clock for the master and slave
NAND flash controller	nand_x_clk	NAND high-speed master and slave clock
	nand_clk	NAND flash clock
OSC1 timer 0	osc1_clk	Clock for the OSC1 timer 0
OSC1 timer 1	osc1_clk	Clock for the OSC1 timer 1
SP timer 0	l4_sp_clk	Clock for the SP timer 0
SP timer 1	l4_sp_clk	Clock for the SP timer 1
I^2C controller 0	l4_sp_clk	Clock for the I^2C 0
I^2C controller 1	l4_sp_clk	Clock for the I^2C 1
I^2C controller 2	l4_sp_clk	Clock for the I^2C 2
I^2C controller 3	l4_sp_clk	Clock for the I^2C 3
UART controller 0	l4_sp_clk	Clock for the UART 0
UART controller 1	l4_sp_clk	Clock for the UART 1
CAN controller 0	l4_sp_clk	Clock for the slave
	can0_clk	CAN 0 controller clock
CAN controller 1	l4_sp_clk	Clock for the slave
	can1_clk	CAN 1controller clock

Table 2–9. Clock Usage By Module (Part 3 of 3)

Module Name	System Clock Name	Use
GPIO interface 0	l4_mp_clk	Clock for the slave
	gpio_db_clk	Debounce clock
GPIO interface 1	l4_mp_clk	Clock for the slave
	gpio_db_clk	Debounce clock
GPIO interface 2	l4_mp_clk	Clock for the slave
	gpio_db_clk	Debounce clock
System manager	osc1_clk	Clock for the system manager
SDRAM subsystem	l4_sp_clk	Clock for the control slave
	ddr_dq_clk	Off-chip data clock
	ddr_dqs_clk	Clock for the MPFE, single-port controller, CSRs, and PHY
	ddr_2x_dqs_clk	Off-chip strobe data clock
	mpu_l2_ram_clk	Clock for the slave connected to MPU subsystem L2 cache
	l3_main_clk	Clock for the slave connected to L3 interconnect
L4 watchdog timer 0	osc1_clk	Clock for the L4 watchdog timer 0
L4 watchdog timer 1	osc1_clk	Clock for the L4 watchdog timer 1
SPI master controller 0	spi_m_clk	Clock for the SPI master 0
SPI master controller 1	spi_m_clk	Clock for the SPI master 1
SPI slave controller 0	l4_main_clk	Clock for the SPI slave 0
SPI slave controller 1	l4_main_clk	Clock for the SPI slave 1
Debug subsystem	l4_mp_clk	System bus clock
	dbg_clk	Debug clock
	dbg_at_clk	Trace bus clock
	dbg_trace_clk	Trace port clock
Reset manager	osc1_clk	Clock for the reset manager
	l4_sp_clk	Clock for the slave
Scan manager	spi_m_clk	Clock for the scan manager
Timestamp generator	dbg_timer_clk	Clock for the timestamp generator

Clock Manager Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for the following module instance:

- **clkmgr**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 2–10 shows the revision history for this document.

Table 2–10. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	■ Reorganized and expanded functional description section. ■ Added address map and register definitions section.
January 2012	1.0	Initial release.

3. Reset Manager

cv_54003-1.2

This chapter provides an overview of HPS reset manager. 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 three reset domains. Each reset domain can be reset independently. Each register in the HPS that can be reset belongs to one particular reset domain.

Table 3–1 shows the HPS reset domains.

Table 3–1. 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 nondebug logic in the FPGA fabric connected to the HPS reset signals.

The HPS supports the following reset types:

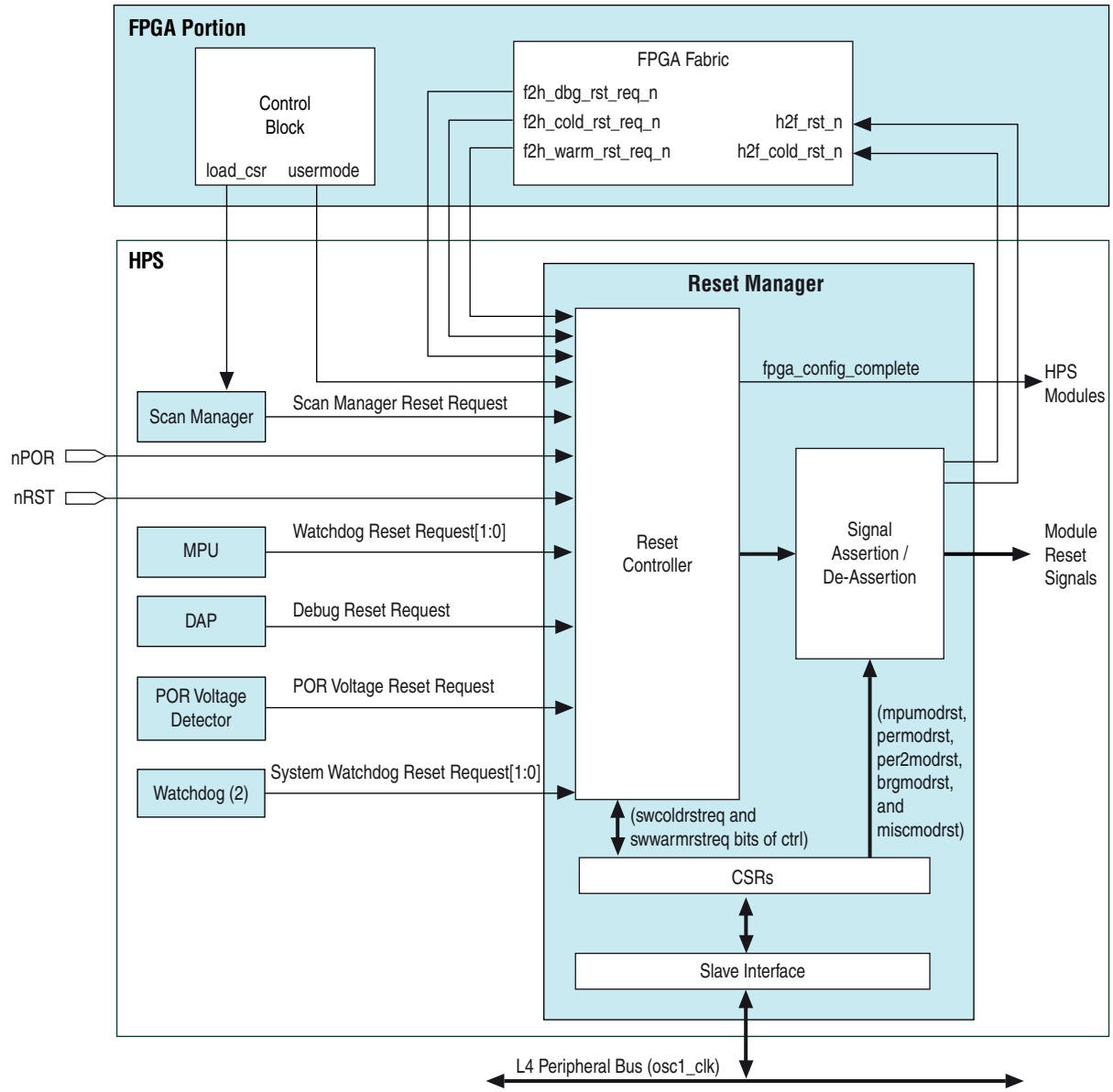
- Cold reset (power-on reset)
  - Used to ensure the HPS is placed in a default state sufficient for software to boot
  - Triggered by a power-on reset and other sources
  - Resets all HPS logic that can be reset
  - Affects all reset domains
- Warm reset
  - Occurs after HPS has already been through 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
  - Occurs after HPS has already been through a cold reset
  - Used to recover debug logic from a non-responsive condition
  - Only affects the debug reset domain

Reset Manager Block Diagram and System Integration

Figure 3–1 shows a block diagram of the reset manager in the SoC device.

Figure 3–1. Reset Manager Block Diagram



Note to Figure 3–1:

(1) Reset-related handshaking signals to other HPS modules and to the clock manager module are omitted from this figure for clarity.

HPS External Reset Sources

Table 3–2 describes the reset sources external to the HPS. All signals are synchronous to the `osc1_clk` clock.

Table 3–2. 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)
`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)

The reset signals from the HPS to the FPGA fabric must be synchronized to your user logic clock domain.

Reset Controller

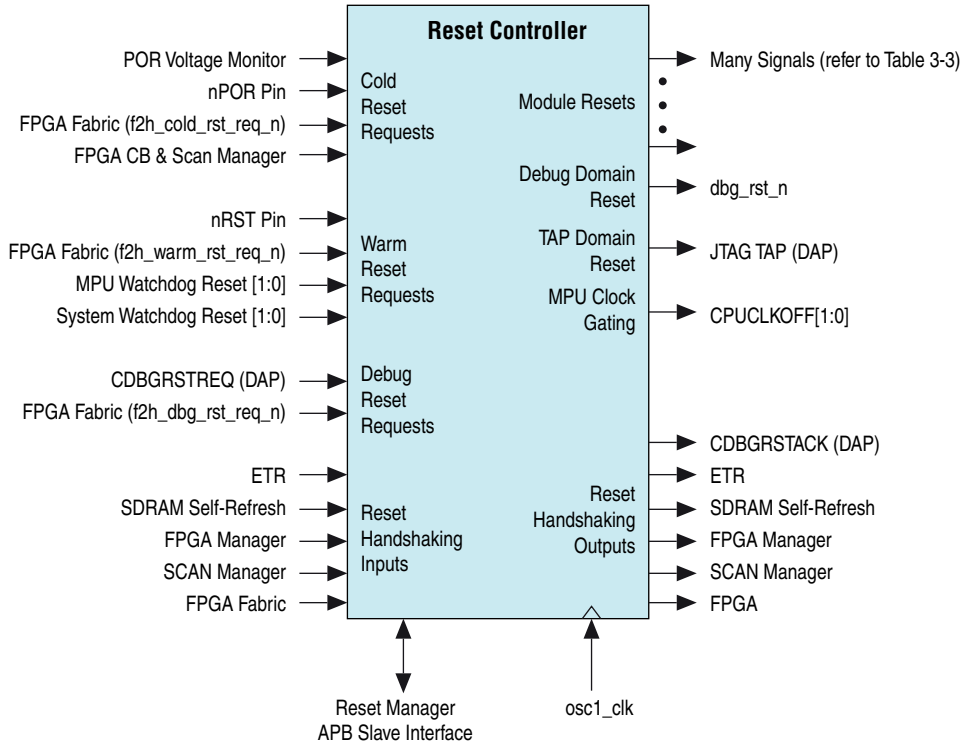
The reset controller performs the following functions:

- 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.

Figure 3–2 shows the reset controller signals.

Figure 3–2. Reset Controller Signals



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 (swwarmrstreq) 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:

- CDBGRSTREQ from DAP
- FPGA fabric

Module Reset Signals

Table 3–3 lists the module reset signals. The module reset signals are organized in groups for the MPU, peripherals, bridges, the level 3 (L3) interconnect, and miscellaneous.

Checkmarks in the Cold Reset, Warm Reset, and Debug Reset columns denote reset signals asserted by each type of reset. For example, writing a 1 to the `swwarmrstreq` bit in the `ctrl` register resets all the modules that have a checkmark in the Warm Reset column.

Checkmarks in the Software Deassert column denote reset signals that are left asserted by the reset manager. To activate the related modules, software can deassert these reset signals as needed by writing to the following reset manager registers:

- MPU module reset register (`mpumodrst`)
- Peripheral module reset register (`permodrst`)
- Peripheral 2 module reset register (`per2modrst`)
- Bridge module reset register (`brgmodrst`)

Table 3–3. Generated Module Resets (Part 1 of 3)

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

Table 3–3. Generated Module Resets (Part 2 of 3)

Group	Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
PER	emac_rst_n[1:0]	Resets each EMAC	System	✓	✓		✓
	usb_rst_n[1:0]	Resets each USB	System	✓	✓		✓
	nand_flash_rst_n	Resets NAND flash controller	System	✓	✓		✓
	qspi_flash_rst_n	Resets quad SPI flash controller	System	✓	✓		✓
	watchdog_rst_n[1:0]	Resets each system watchdog timer	System	✓	✓		✓
	osc1_timer_rst_n[1:0]	Resets each OSC1 timer	System	✓	✓		✓
	sp_timer_rst_n[1:0]	Resets each SP timer	System	✓	✓		✓
	i2c_rst_n[3:0]	Resets each I^2C controller	System	✓	✓		✓
	uart_rst_n[1:0]	Resets each UART	System	✓	✓		✓
	spim_rst_n[1:0]	Resets SPI master controller	System	✓	✓		✓
	spis_rst_n[1:0]	Resets SPI slave controller	System	✓	✓		✓
	sdmmc_rst_n	Resets SD/MMC controller	System	✓	✓		✓
	can_rst_n[1:0]	Resets each CAN controller	System	✓	✓		✓
	gpio_rst_n[2:0]	Resets each GPIO interface	System	✓	✓		✓
	dma_rst_n	Resets DMA controller	System	✓	✓		✓
	sdram_rst_n	Resets SDRAM subsystem (resets logic associated with cold or warm reset)	System	✓	✓		✓
PER2	dma_periph_if_rst_n[7:0]	DMA controller request interface from FPGA fabric to DMA controller	System	✓	✓		✓
Bridge	hps2fpga_bridge_rst_n	Resets HPS-to-FPGA AMBA® Advanced eXtensible Interface (AXI™) bridge	System	✓	✓		✓
	fpga2hps_bridge_rst_n	Resets FPGA-to-HPS AXI bridge	System	✓	✓		✓
	lwhps2fpga_bridge_rst_n	Resets lightweight HPS-to-FPGA AXI bridge	System	✓	✓		✓

Table 3–3. Generated Module Resets (Part 3 of 3)

Group	Module Reset Signal	Description	Reset Domain	Cold Reset	Warm Reset	Debug Reset	Software Deassert
MISC	boot_rom_rst_n	Resets boot ROM	System	✓	✓		
	onchip_ram_rst_n	Resets on-chip RAM	System	✓	✓		
	sys_manager_rst_n	Resets system manager (resets logic associated with cold or warm reset)	System	✓	✓		
	sys_manager_cold_rst_n	Resets system manager (resets logic associated with cold reset only)	System	✓			
	fpga_manager_rst_n	Resets FPGA manager	System	✓	✓		
	acp_id_mapper_rst_n	Resets ACP ID mapper	System	✓	✓		
	h2f_rst_n	Resets user logic in FPGA fabric (resets logic associated with cold or warm reset)	System	✓	✓		
	h2f_cold_rst_n	Resets user logic in FPGA fabric (resets logic associated with cold reset only)	System	✓			
	rst_pin_rst_n	Pulls nRST pin low	System	✓	✓		
	timestamp_cold_rst_n	Resets debug timestamp to 0x0	System	✓			
	clk_manager_cold_rst_n	Resets clock manager (resets logic associated with cold reset only)	System	✓			
	scan_manager_rst_n	Resets scan manager	System	✓	✓		
	frz_ctrl_cold_rst_n	Resets freeze controller (resets logic associated with cold reset only)	System	✓			
	sys_dbg_rst_n	Resets debug masters and slaves connected to L3 interconnect and level 4 (L4) buses	System	✓	✓		
	dbg_rst_n	Resets debug components including DAP, trace, MPU debug logic, and any user debug logic in the FPGA fabric	Debug	✓		✓	
	tap_cold_rst_n	Resets portion of TAP controller in the DAP that must be reset on a cold reset	TAP	✓			
	sdram_cold_rst_n	Resets SDRAM subsystem (resets logic associated with cold reset only)	System	✓			
L3	l3_rst_n	Resets L3 interconnect and L4 buses	System	✓	✓		

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 reset request. The stat register captures all reset requests that have occurred. Software is responsible for clearing the bits.

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.

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

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.

Reset Sequencing

The reset controller sequences resets without software assistance. Module reset signals are asserted asynchronously at the same time. 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.

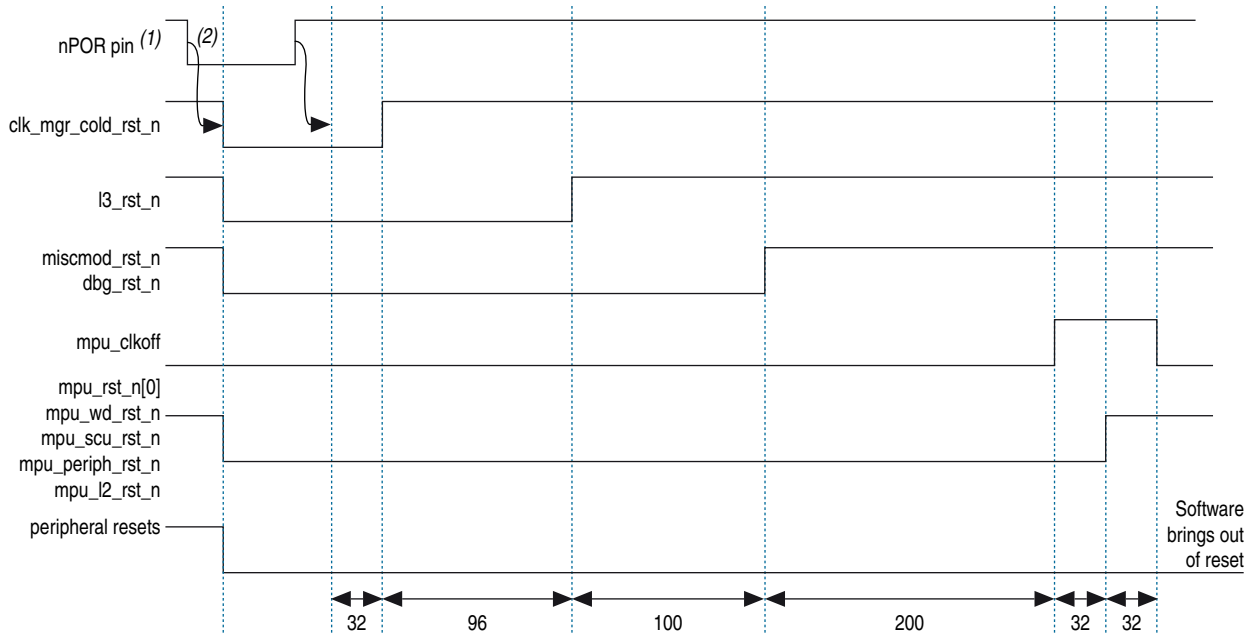
For information about safe mode options, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 Table 3–3. 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.

Figure 3–3 shows the timing diagram for cold reset.

Figure 3–3. Cold Reset Timing Diagram



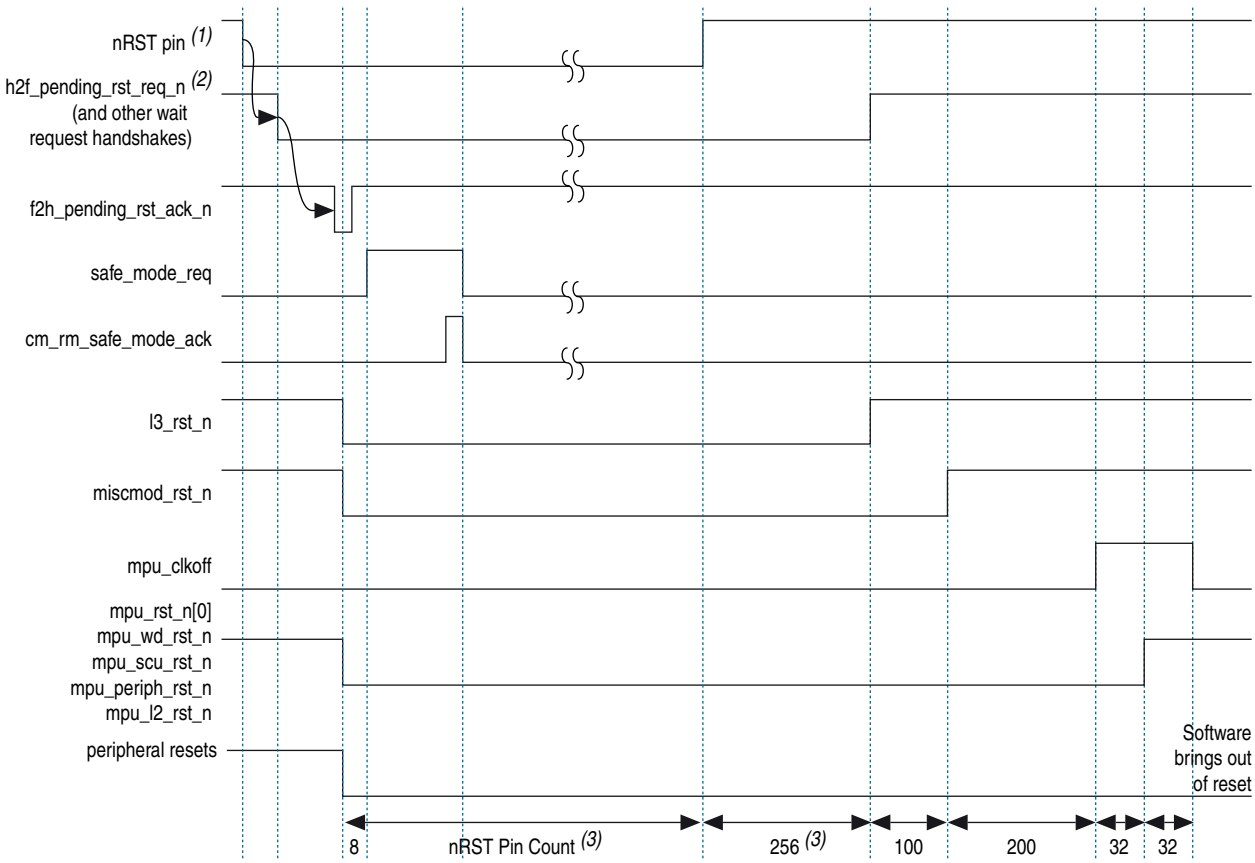
Note to Figure 3–3:

(1) Cold reset can be initiated from several other sources. For the complete list, refer to "Reset Controller" on page 3–3.

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

Figure 3–4 shows the timing diagram for warm reset.

Figure 3–4. Warm Reset Timing Diagram



Notes to Figure 3–4:

(1) Warm reset can be initiated from several other sources. For the complete list, refer to "Reset Controller" on page 3–3.

(2) For information about the wait request reset handshaking, refer to "Reset Handshaking" on page 3–13.

(3) When the nRST pin 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.

Cold Reset Assertion Sequence

The following list describes the assertion steps for cold reset shown in Figure 3–3:

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".

Warm Reset Assertion Sequence

The following list describes the assertion steps for warm reset shown in Figure 3–4:

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".

Cold and Warm Reset Deassertion Sequence

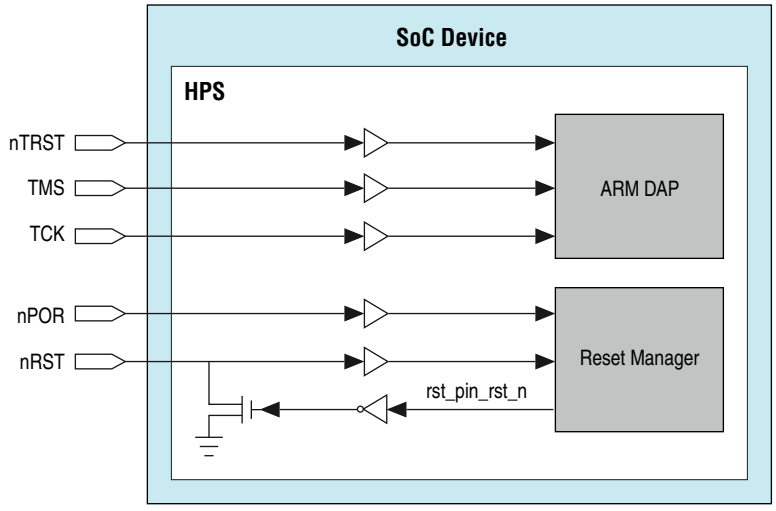
The following list describes the deassertion steps for both cold and warm reset shown in Figure 3–3 and Figure 3–4:

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 3–5 shows all HPS pins related to reset.

Figure 3–5. 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. Any warm reset request causes the reset manager to drive the rst_pin_rst_n signal output low, which 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.

For more information, refer to the individual chapters in volume 3 of the *Cyclone V Device Handbook*.

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.

Altera strongly recommends using Altera-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, Altera recommends to only disable safe mode 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 puts 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.

For more information about safe mode, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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:

- 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 reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for any of the following module instances:

- **rstmgr**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 3–4 shows the revision history for this document.

Table 3–4. Document Revision History

Date	Version	Changes
November 2012	1.2	■ Added cold and warm reset timing diagrams. ■ Minor updates.
May 2012	1.1	Added reset controller, functional description, and address map and register definitions sections.
January 2012	1.0	Initial release.

Section II. System Interconnect

This section includes the following chapters:

- Chapter 4, Interconnect
- Chapter 5, HPS-FPGA AXI Bridges

For information about the revision history for chapters in this section, refer to "Document Revision History" in each individual chapter.

4. Interconnect

cv_54004-1.2

The hard processor system (HPS) level 3 (L3) interconnect and level 4 (L4) peripheral buses are implemented with the ARM® CoreLink™ Network Interconnect (NIC-301). The NIC-301 provides a foundation for a high-performance HPS 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 L3 interconnect implements a multilayer, nonblocking architecture that supports multiple simultaneous transactions between masters and slaves, including the Cortex™-A9 microprocessor unit (MPU) subsystem. The interconnect provides five independent L4 buses to access control and status registers (CSRs) of peripherals, managers, and memory controllers

Additional information is available in the *AMBA Network Interconnect (NIC-301) Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

Features of the Interconnect

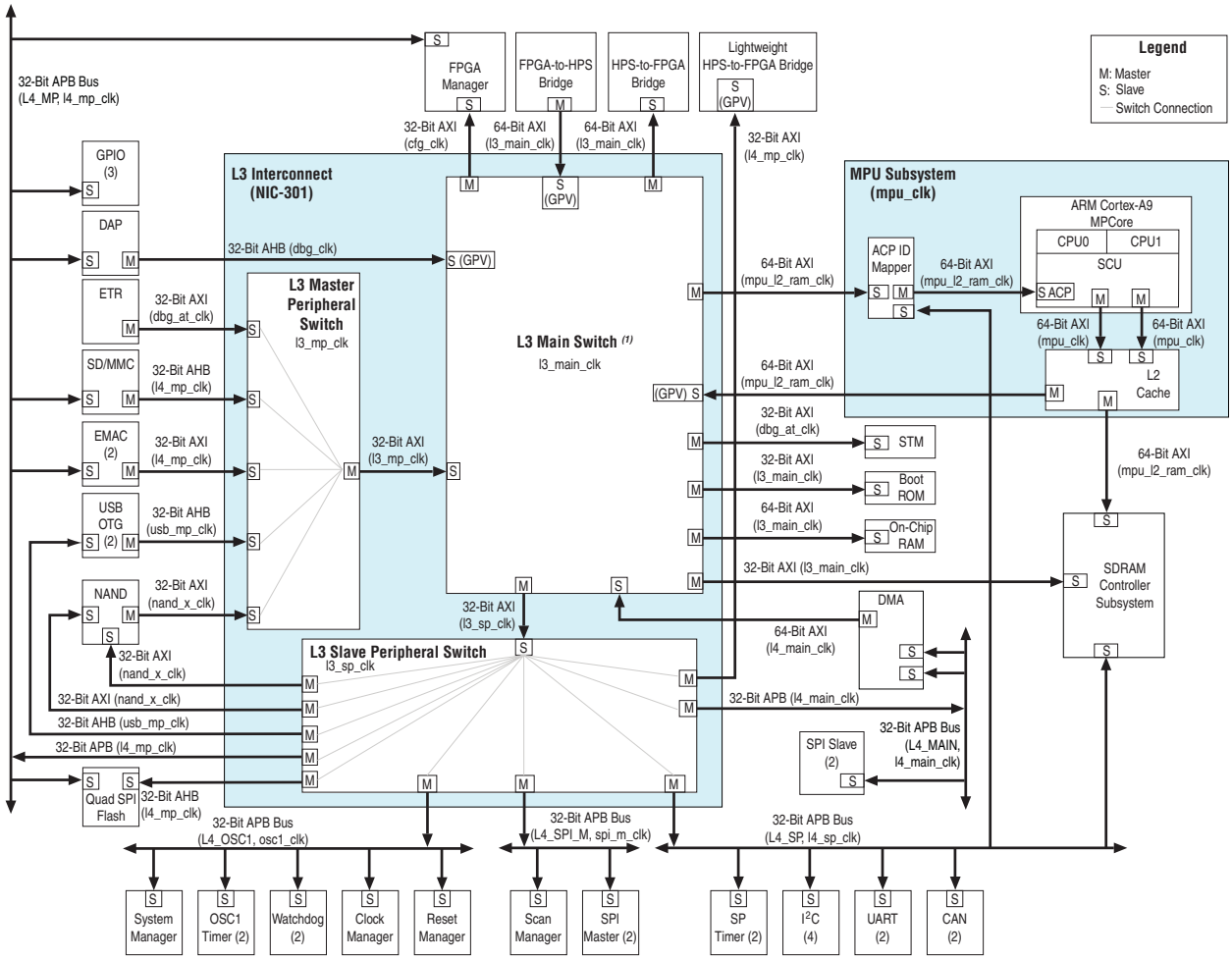
The L3 interconnect has the following characteristics:

- 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
- Security of the following types:
  - Secure
  - Nonsecure
  - Per transaction security
- Five independent L4 buses

Interconnect Block Diagram and System Integration

Figure 4–1 shows a block diagram of the L3 interconnect and L4 buses.

Figure 4–1. Interconnect Block Diagram



Note to Figure 4–1:

(1) For L3 main switch connection details, refer to Table 4–1.

The L3 interconnect is a partially-connected switch fabric; not all masters can access all slaves. For more information, refer to "Master-to-Slave Connectivity Matrix" on page 4–6.

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

- L3 main switch
  - Main switch 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 main switch
  - 32-bit data width
  - Operates at up to half the main switch clock frequency
- L3 slave peripheral switch
  - Used to provide access to level 3 and 4 slave interfaces for masters of the master peripheral and main switches
  - 32-bit data width
  - Five independent L4 buses

The L3 master and slave peripheral switches are fully-connected crossbars. The L3 main switch is a partially-connected crossbar. Table 4–1 shows the connectivity matrix of all the master and slave interfaces of the L3 main switch. Checkmarks denote connections.

Table 4–1. L3 Main Switch Connectivity Matrix

Masters	Slaves							
	L3 Slave Peripheral Switch	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			✓	✓			✓	✓
L2 Cache Master 0	✓	✓	✓		✓	✓	✓	
FPGA-to-HPS Bridge	✓			✓	✓		✓	✓
DMA	✓	✓	✓	✓	✓		✓	✓
DAP	✓	✓	✓	✓			✓	✓

L3 Masters

The following list contains all of the master interfaces connected to the L3 interconnect:

- MPU subsystem—L2 cache master 0 connected to the L3 main switch
- FPGA-to-HPS bridge—Connected to the L3 main switch
- DMA—Connected to the L3 main switch
- EMAC0—Connected to the L3 master peripheral switch
- EMAC1—Connected to the L3 master peripheral switch
- USB0—Connected to the L3 master peripheral switch
- USB1—Connected to the L3 master peripheral switch
- NAND—Connected to the L3 master peripheral switch
- SD/MMC—Connected to the L3 master peripheral switch

- ETR—Connected to the L3 master peripheral switch
- DAP—Connected to the L3 main switch

L3 Slaves

The following list contains all of the slave interfaces connected to the L3 interconnect:

- USB0—CSR slave interface connected to the L3 slave peripheral switch
- USB1—CSR slave interface connected to the L3 slave peripheral switch
- NAND registers—CSR slave interface connected to the L3 slave peripheral switch
- NAND data—Command and data slave interface connected to the L3 slave peripheral switch
- Quad SPI flash—Data slave interface connected to the L3 slave peripheral switch
- FPGA manager—Data slave interface connected to the L3 main switch
- HPS-to-FPGA bridge—Data slave interface connected to the L3 main switch
- Lightweight HPS-to-FPGA bridge—Data slave interface connected to the L3 slave peripheral switch
- ACP ID mapper—Data slave interface connected to the L3 main switch
- STM—Connected to the L3 main switch
- Boot ROM—Connected to the L3 main switch
- On-chip RAM—Connected to the L3 main switch
- SDRAM controller subsystem—SDRAM multi-port front end slave interface connected to the L3 main switch

L4 Slaves

Each of the L4 slaves is an APB slave connected to one of the five following L4 buses:

- L4 slave peripheral (SP) bus—APB for peripherals that do not require fast access
  - SDRAM controller subsystem—CSR access
  - SP timer 0—CSR access
  - SP timer 1—CSR access
  - I2C0—CSR access
  - I2C1—CSR access
  - I2C2 (associated with EMAC0)—CSR access
  - I2C3 (associated with EMAC1)—CSR access
  - UART0—CSR access
  - UART1—CSR access
  - CAN0—CSR access
  - CAN1—CSR access

- L4 master peripheral (MP) bus—APB that provides access to primarily the L3 master peripherals.
  - ACP ID mapper—CSR access
  - FPGA manager—CSR access
  - DAP—CSR access
  - Quad SPI flash—CSR access
  - SD/MMC—CSR access
  - EMAC0—CSR access
  - EMAC1—CSR access
  - GPIO0—CSR access
  - GPIO1—CSR access
  - GPIO2—CSR access
- L4 oscillator 1 (OSC1) bus—APB dedicated to peripherals that operate on the external oscillator 1 domain.
  - OSC1 timer 0—CSR access
  - OSC1 timer 1—CSR access
  - Watchdog 0—CSR access
  - Watchdog 1—CSR access
  - Clock manager—CSR access
  - Reset manager—CSR access
  - System manager—CSR access
- L4 main bus—APB dedicated to the DMA and SPI slaves
  - DMA_s—Access to the DMA controllers secure registers
  - DMA_ns—Nonsecure access to the DMA controller nonsecure registers
  - SPI slave 0—CSR access
  - SPI slave 1—CSR access
- L4 SPI master (SPIM) bus—APB dedicated to the SPI masters and scan manager.
  - SPI master 0—CSR access
  - SPI master 1—CSR access
  - Scan manager—CSR access

Functional Description of the Interconnect

This section provides a functional description of the interconnect.

Master-to-Slave Connectivity Matrix

The interconnect is a partially-connected crossbar. Table 4–2 shows the connectivity matrix of all the master and slave interfaces of the interconnect. Checkmarks denote connections.

Table 4–2. 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	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓

Address Remapping

The interconnect supports address remapping through the `remap` register. 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 and maps into the address space of the following L3 masters:

- 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 interconnect.

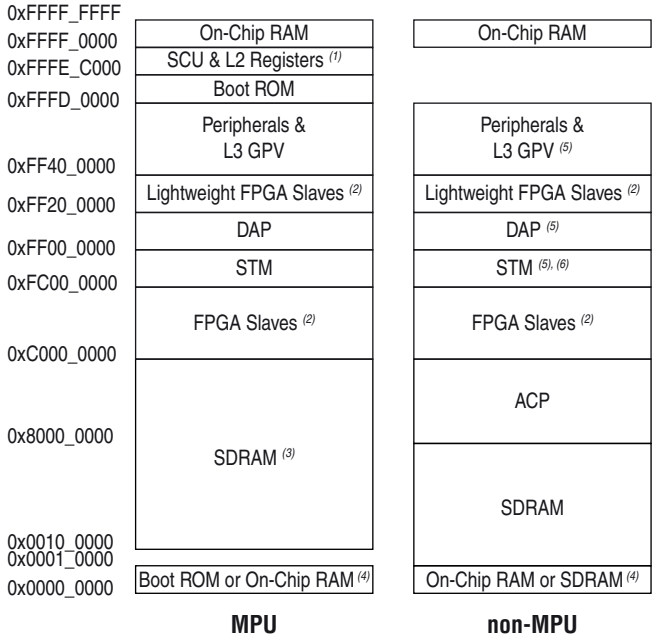
The effects of the remap bits can be categorized in the following groups:

- 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

Figure 4–2 shows the interconnect address map for all MPU and non-MPU masters. The figure is not to scale.

Figure 4–2. Address Map per Master



Notes to Figure 4–2:

(1) The SCU and L2 cache registers are located in the MPU subsystem and are not accessible from the L3 interconnect.

(2) This address range is not always accessible. For more information, refer to Table 4–3.

(3) The MPU subsystem has one master that connects to the interconnect and another master that connects directly to the SDRAM controller subsystem. The address filter registers in the MPU L2 control which MPU addresses are sent to each master. This figure assumes the filter registers contain their reset values.

(4) This address range is configurable. For more information, refer to Table 4–3.

(5) This address range is not accessible from the master peripheral interfaces. For more information, refer to "Master-to-Slave Connectivity Matrix" on page 4–6.

(6) This address range is not accessible from the DAP interface. For more information, refer to "Master-to-Slave Connectivity Matrix" on page 4–6.

For the MPU L3 master, either the boot ROM or on-chip RAM maps to address 0x0 and obscures the lowest 64 K of SDRAM. The address space from 0x0001_0000 to 0x0010_0000 is not accessible because the MPU L2 filter registers 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 K of SDRAM for non-MPU masters.

Table 4–3 lists how the remap bits affect the memory maps.

Table 4–3. Memory Map Remap Bits

Bit Name	Bit Offset	Description
mpuzero	0	When set to 0, the boot ROM maps to address 0x0 for the MPU L3 master. When set to 1, the on-chip RAM maps to address 0x0 for the MPU L3 master. This bit has no effect on non-MPU masters. Note that regardless of this setting, the boot ROM also always maps to address 0xfffd_0000 and the on-chip RAM also always maps to address 0xffff_0000 for the MPU L3 master.
nonmpuzero	1	When set to 0, the SDRAM maps to address 0x0 for the non-MPU L3 masters. When set to 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 0xffff_0000 for the non-MPU L3 masters.
Reserved	2	Must always be set to 0.
hps2fpga	3	When set to 1, the HPS-to-FPGA bridge slave port is visible to the L3 masters. When set to 0, accesses to the associated address range return an AXI decode error to the master.
lwhp2fpga	4	When set to 1, the lightweight HPS-to-FPGA bridge slave port is visible to the L3 masters. When set to 0, accesses to the associated address range return an AXI decode error to the master.
Reserved	31:5	Must always be set to 0.

L2 filter registers in the MPU subsystem, not the interconnect, allow the SDRAM to be remapped to address 0 for the MPU. For more information about the MPU subsystem, refer to the *Cortex-A9 MPU System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Master Caching and Buffering Overrides

Some of the masters of the interconnect do not have the ability to drive the caching and buffering signals of their AXI and AHB interfaces. In order to ensure that these masters can perform transfers efficiently, the registers are available from the system manager so that you can enable cacheable and bufferable transactions. The following masters have their caching and buffering signals driven by the system manager:

- EMAC0 and EMAC1
- USB OTG 0 and USB OTG 1
- NAND flash
- SD/MMC

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 interconnect, so avoid changing these settings when any of the masters are active. For more information about enabling or disabling this feature, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 interconnect are either secure, nonsecure, or the security is set on a per transaction basis. The DAP is capable of performing only 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 interconnect masters perform nonsecure accesses. For more information, refer to "Interconnect Master Properties" on page 4–10.

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

Arbitration

At the entry point to the interconnect, all transactions are allocated a local quality of service (QoS) value that you can programmatically configure. The arbitration of the transaction throughout the infrastructure uses this QoS value. The QoS controls for each master connected to the interconnect are separated into read and write QoS priority values.

At any arbitration node, a fixed priority exists for transactions with different QoS values. The highest QoS value has the highest priority. If there are coincident transactions at an arbitration node with the same QoS value that require arbitration, then the interconnect uses a least recently used (LRU) algorithm.

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. The CDAS implementation for the masters is described in"Interconnect Master Properties" on page 4–10.

The following schemes are available:

- Single Slave
- Single Slave Per ID
- Single Active Slave

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.

Interconnect Master Properties

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

Table 4–4 shows all the master interfaces connected to the interconnect.

Table 4–4. Interconnect Master Interfaces (Part 1 of 2)

Master	Interface Width	Clock	Switch	Security	GPV Access	CDAS	Issuance (1)	Buffer Depth (2)	Type
L2 cache M0	64	mpu_l2_ram_clk	L3 main switch	Per Transaction	Yes	SSPID	7, 12, 19	2, 2, 2, 2, 2	AXI
FPGA-to-HPS bridge	64	l3_main_clk	L3 main switch	Per Transaction	Yes	SAS	16, 16, 32	2, 2, 6, 6, 2	AXI

Table 4–4. Interconnect Master Interfaces (Part 2 of 2)

Master	Interface Width	Clock	Switch	Security	GPV Access	CDAS	Issuance (1)	Buffer Depth (2)	Type
DMA	64	l4_main_clk	L3 main switch	Per Transaction	No	SSPID	8, 8, 8	2, 2, 2, 2, 2	AXI
EMAC 0/1	32	l4_main_clk	L3 master peripheral switch	Nonsecure	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	Nonsecure	No	SSPID	32, 1, 32	2, 2, 2, 2, 2	AXI
DAP	32	dbg_clk	L3 main switch	Secure	Yes	SS	1, 1, 1	2, 2, 2	AHB

Notes to Table 4–4:

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

(2) 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.

Interconnect Slave Properties

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

Table 4–5 shows all the slave interfaces connected to the interconnect.

Table 4–5. Interconnect Slave Interfaces (Part 1 of 3)

Slave	Interface Width	Clock	Mastered By	Acceptance (1)	Buffer Depth (2)	Interface 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^2C 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
CAN 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

Table 4–5. Interconnect Slave Interfaces (Part 2 of 3)

Slave	Interface Width	Clock	Mastered By	Acceptance (1)	Buffer Depth (2)	Interface Type
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 SP 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
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
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 main switch	1, 2, 3	2, 2, 2, 32, 2	AXI
HPS-to-FPGA bridge	64	l3_main_clk	L3 main switch	16, 16, 32	2, 2, 6, 6, 2	AXI
ACP ID mapper data	64	mpu_l2_ram_clk	L3 main switch	13, 5, 18	2, 2, 2, 2, 2	AXI
STM	32	dbg_at_clk	L3 main switch	1, 2, 2	2, 2, 2, 2, 2	AXI
On-chip boot ROM	32	l3_main_clk	L3 main switch	1, 1, 2	0, 0, 0, 0, 0	AXI
On-chip RAM	64	l3_main_clk	L3 main switch	2, 2, 2	0, 0, 0, 8, 0	AXI

Table 4–5. Interconnect Slave Interfaces (Part 3 of 3)

Slave	Interface Width	Clock	Mastered By	Acceptance (1)	Buffer Depth (2)	Interface Type
SDRAM subsystem L3 data	32	l3_main_clk	L3 main switch	16, 16, 16	2, 2, 2, 2, 2	AXI

Notes to Table 4–5:

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

(2) 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 the W, A, and D channels.

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

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.

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 (infocenter.arm.com).

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

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

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. For more information, refer to"Upsizing Data Width Function" on page 4–13.

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

If the programmable bit bypass_merge 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. Bypass merge is accessible through the GPV registers and is only accessible to secure masters.

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, described in "Upsizing Data Width Function" on page 4–13.

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.

For more information about exclusive access support, refer to the *SDRAM Controller Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*.

FIFO Buffers and Clocks

The interconnect contains FIFO buffers in the majority of the interfaces exposed to the HPS master and slaves, as well as between the subswitches. These FIFO buffers also provide clock domain crossing for masters and slaves that operate at a different clock frequency than the switch they connect to.

Data Release Mechanism

For network ports containing write data FIFO buffers with a depth of four or greater, 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 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.

For more information about which interfaces contain write data FIFO buffers with a depth of four or greater, refer to "Interconnect Master Properties" on page 4–10.

Resets

The interconnect has one reset signal. The reset manager drives this signal to the SD/MMC controller on a cold or warm reset. On reset, the boot ROM is mapped to address 0x0. The DAP virtually maps to ID 2.

For more information about resets, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*. For information about virtual ID mapping by the ACP ID mapper, refer to the *Cortex-A9 MPU System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Interconnect Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for the following module instance:

- **l3regs**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 4–6 shows the revision history for this document.

Table 4–6. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
June 2012	1.1	■ Added main switch connectivity matrix table. ■ Rearranged functional description sections. ■ Simplified address remapping section. ■ Added address map and register definitions section.
January 2012	1.0	Initial release.

5. HPS-FPGA AXI Bridges

cv_54005-1.1

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).

Additional information is available in the *AMBA AXI Protocol Specification v1.0* and the *AMBA Network Interconnect (NIC-301) Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

The HPS contains the following HPS-FPGA AXI bridges:

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

Features of the AXI Bridges

The HPS-FPGA AXI bridges allow masters in the FPGA fabric to communicate with slaves in the HPS logic and vice versa. For example, you can instantiate additional memories or peripherals in the FPGA fabric, and master interfaces belonging to components in the HPS logic can access them. You can also instantiate components such as a Nios® II processor in the FPGA fabric and their master interfaces can access memories or peripherals in the HPS logic.

The AXI bridges provide the features listed in Table 5–1.

Table 5–1. AXI Bridge Features

Feature	FPGA-to-HPS Bridge	HPS-to-FPGA Bridge	Lightweight HPS-to-FPGA Bridge
Supports the AMBA AXI3 interface protocol	✓	✓	✓
Implements clock crossing and manages the transfer of data across the clock domains in the HPS logic and the FPGA fabric	✓	✓	✓
Performs data width conversion between the HPS logic and the FPGA fabric	✓	✓	✓
Allows configuration of FPGA interface widths at instantiation time	✓	✓	

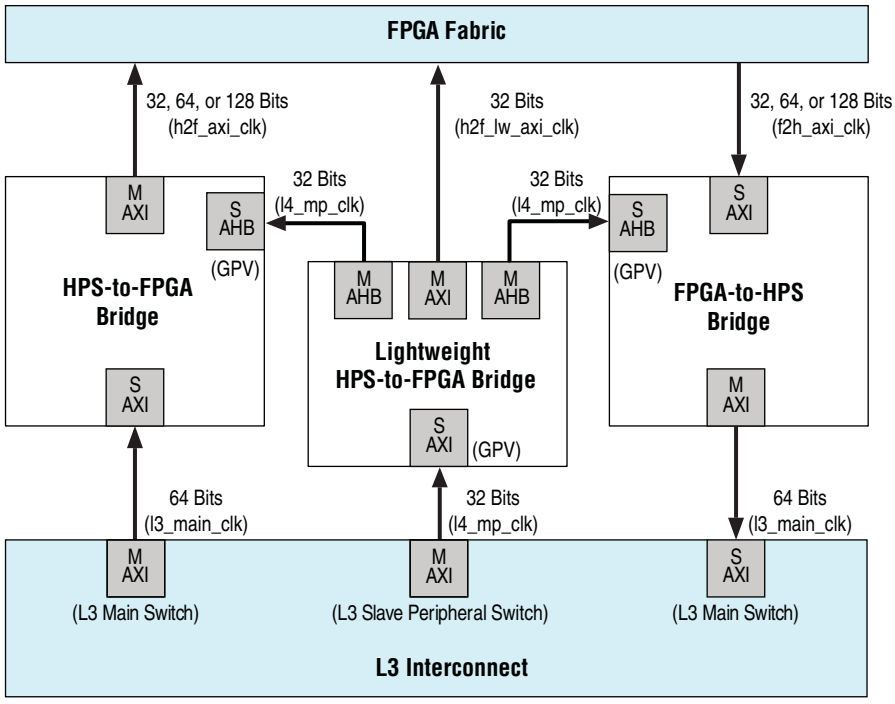
Each bridge consists of an AXI master-slave pair with one interface exposed to the FPGA fabric and the other exposed to the HPS logic. The HPS-to-FPGA and lightweight HPS-to-FPGA bridges expose an AXI master interface that you can connect to AXI or Avalon-MM slave interfaces in the FPGA fabric. The FPGA-to-HPS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

For information about configuring the AXI bridges, refer to the *Instantiating the HPS Component* chapter in volume 3 of the *Cyclone V Device Handbook*.

AXI Bridges Block Diagram and System Integration

Figure 5–1 shows a block diagram of the AXI 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 clock domains are described in "Clocks and Resets" on page 5–12.

Figure 5–1. AXI Bridge Connectivity



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.

For more information about connectivity, such as which masters have access to each bridge, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

Functional Description of the AXI 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.

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 5–2 lists the properties of the FPGA-to-HPS bridge, including the configurable slave interface exposed to the FPGA fabric.

Table 5–2. FPGA-to-HPS Bridge Properties

Bridge Property	FPGA Slave Interface	L3 Master Interface
Data width (1)	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

Note to Table 5–2:

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

The FPGA-to-HPS bridge contains a GPV, described in "The Global Programmers View". 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.

It is critical to provide the correct l4_mp_clk clock to support access to the GPV, as described in "GPV Clocks" on page 5–13.

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.

Table 5–3 lists some possible master and FPGA-to-HPS bridge slave configurations that support accesses to the L2 cache with ECC enabled.

Table 5–3. FPGA Master and FPGA-to-HPS Bridge Configurations

Soft Logic Master Width	Soft Logic Master Alignment	Soft Logic Master Burst Size (Width)	Soft Logic Master Burst Length	FPGA-to-HPS Bridge Slave Width
32 bits	8 bytes	4 bytes	2, 4, 6, 8, 10, 12, 14, or 16 beats	32 bits
64 bits	8 bytes	8 bytes	1 to 16 beats	32 bits
128 bits	8 or 16 bytes	8 or 16 bytes	1 to 16 beats	32 bits
32 bits	8 bytes	4 bytes	2, 4, 6, 8, 10, 12, 14, or 16 beats	64 bits
64 bits	8 bytes	8 bytes	1 to 16 beats	64 bits
128 bits	8 or 16 bytes	8 or 16 bytes	1 to 16 beats	64 bits
32 bits	8 bytes	4 bytes	2, 4, 6, 8, 10, 12, 14, or 16 beats	128 bits
64 bits	8 bytes	8 bytes	1 to 16 beats	128 bits
128 bits	8 or 16 bytes	8 or 16 bytes	1 to 16 beats	128 bits

For more information about the ECC option of the L2 cache, refer to the *Cortex-A9 Microprocessor Unit Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*.

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. Table 5–4 through Table 5–8 list all the signals exposed by the FPGA-to-HPS slave interface to the FPGA fabric.

Table 5–4 lists the slave write address channel signals.

Table 5–4. 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

Table 5–4. FPGA-to-HPS Bridge Slave Write Address Channel Signals

Signal	Width	Direction	Description
AWVALID	1 bit	Input	Write address channel valid
AWREADY	1 bit	Output	Write address channel ready
AWUSER	5 bits	Input	User sideband signals

Table 5–5 lists the slave write data channel signals.

Table 5–5. FPGA-to-HPS Bridge Slave Write Data Channel Signals

Signal	Width	Direction	Description
WID	8 bits	Input	Write ID
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 5–6 lists the slave write response channel signals.

Table 5–6. 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 5–7 lists the slave read address channel signals.

Table 5–7. 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
ARUSER	5 bits	Input	Read user sideband signals

Table 5–8 lists the slave read data channel signals.

Table 5–8. 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

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 64 megabytes (MB). The address space size is less than 1 GB because 64 MB is 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.

For detailed information about which masters have access to each bridge, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*. For details about L2 cache address filtering, refer to the *Cortex-A9 Microprocessor Unit Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*.

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.

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.

For more information about enabling the HPS-to-FPGA bridge, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

Table 5–9 lists the properties of the HPS-to-FPGA bridge, including the configurable master interface exposed to the FPGA fabric.

Table 5–9. HPS-to-FPGA Bridge Properties

Bridge Property	L3 Slave Interface	FPGA Master Interface
Data width (1)	64 bits	32, 64, or 128 bits
Clock domain	`l3_main_clk`	`h2f_axi_clk`
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

Note to Table 5–9:

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

The HPS-to-FPGA bridge's GPV, described in "The Global Programmers View" on page 5–3, 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.

It is critical to provide the correct `l4_mp_clk` clock to support access to the GPV, as described in "GPV Clocks" on page 5–13.

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. Table 5–10 through Table 5–14 list all the signals exposed by the HPS-to-FPGA master interface to the FPGA fabric.

Table 5–10 lists the master write address channel signals.

Table 5–10. 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
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 5–11 lists the master write data channel signals.

Table 5–11. 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 5–12 lists the master write response channel signals.

Table 5–12. 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 5–13 lists the master read address channel signals.

Table 5–13. 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 5–14 lists the master read data channel signals.

Table 5–14. 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

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.

For detailed information about which masters have access to each bridge, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 CSR access latency and overall system performance.

Table 5–15 lists the properties of the lightweight HPS-to-FPGA bridge, including the master interface exposed to the FPGA fabric.

Table 5–15. Lightweight HPS-to-FPGA Bridge Properties

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, as shown in Figure 5–1 on page 5–2. 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 GPV to control the behavior of its four interfaces (one slave and three masters). The GPV is described in "The Global Programmers View" on page 5–3.

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.

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.

Lightweight HPS-to-FPGA Bridge Master Signals

All the lightweight HPS-to-FPGA bridge master signals have a fixed width. Table 5–16 through Table 5–20 list all the signals exposed by the lightweight HPS-to-FPGA master interface to the FPGA fabric.

Table 5–16 lists the master write address channel signals.

Table 5–16. 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 5–17 lists the master write data channel signals.

Table 5–17. 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
WREADY	1 bit	Input	Write data channel ready

Table 5–18 lists the master write response channel signals.

Table 5–18. 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 5–19 lists the master read address channel signals.

Table 5–19. 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 5–20 lists the master read data channel signals.

Table 5–20. 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
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

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.

For information about the `f2h_axi_clk` clock, refer to the *HPS Component Interfaces* chapter in volume 3 of the *Cyclone V Device Handbook*. For information about the `l3_main_clk` and `l4_mp_clk` clocks, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

The FPGA-to-HPS bridge has one reset signal, `fpga2hps_bridge_rst_n`. The reset manager drives this signal to FPGA-to-HPS bridge on a cold or warm reset.

For more information about the reset manager, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

HPS-to-FPGA Bridge

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.

For information about the `l3_main_clk` clock, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*. For information about the `h2f_axi_clk` clock, refer to the *HPS Component Interfaces* chapter in volume 3 of the *Cyclone V Device Handbook*.

The HPS-to-FPGA bridge has one reset signal, `hps2fpga_bridge_rst_n`. The reset manager drives this signal to HPS-to-FPGA bridge on a cold or warm reset.

For more information about the reset manager, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Lightweight HPS-to-FPGA Bridge

The master interface into the FPGA fabric operates in the `h2f_lw_axi_clk` clock domain 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.

For information about the `l4_mp_clk` clock, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*. For information about the `h2f_lw_axi_clk` clock, refer to the *HPS Component Interfaces* chapter in volume 3 of the *Cyclone V Device Handbook*.

The lightweight HPS-to-FPGA bridge has one reset signal, `lwhps2fpga_bridge_rst_n`. The reset manager drives this signal to the lightweight HPS-to-FPGA bridge on a cold or warm reset.

For more information about the reset manager, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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. For information about the GPV, refer to "The Global Programmers View" on page 5–3.

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.

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 AXI Bridges Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the following links for the module instance:

- **fpga2hpsregs**
- **hps2fpgaregs**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 5–21 lists the revision history for this document.

Table 5–21. Document Revision History

Date	Version	Changes
November 2012	1.1	Described GPV.
January 2012	1.0	Initial release.

Section III. Cortex-A9 Microprocessor

This section includes the following chapters:

- Chapter 6, Cortex-A9 Microprocessor Unit Subsystem

For information about the revision history for chapters in this section, refer to "Document Revision History" in each individual chapter.

6. Cortex-A9 Microprocessor Unit Subsystem

cv_54006-1.2

The hard processor system (HPS) in the Altera® SoC FPGA device includes a stand-alone, full-featured ARM® Cortex™-A9 MPCore™, single- or dual-core 32-bit application processor. The Cortex-A9 MPU subsystem is composed of a Cortex-A9 MPCore, a level 2 (L2) cache, an Accelerator Coherency Port (ACP) ID mapper, and debugging modules.

Features of the Cortex-A9 MPU Subsystem

The Altera Cortex-A9 MPU subsystem provides the following features:

- One or two Cortex-A9 processors
- Interrupt controller
- Private interval and watchdog timer for each processor
- Global timer
- TrustZone® system security extensions
- Symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP) modes
- Debugging modules

Cortex-A9 MPU Subsystem Block Diagram and System Integration

Figure 6–1 shows a dual-core MPU subsystem in the context of the HPS, with the L2 cache. The L2 cache can access either the level 3 (L3) interconnect fabric or the SDRAM.

Figure 6–1. Cortex-A9 MPU Subsystem with L3 Interconnect

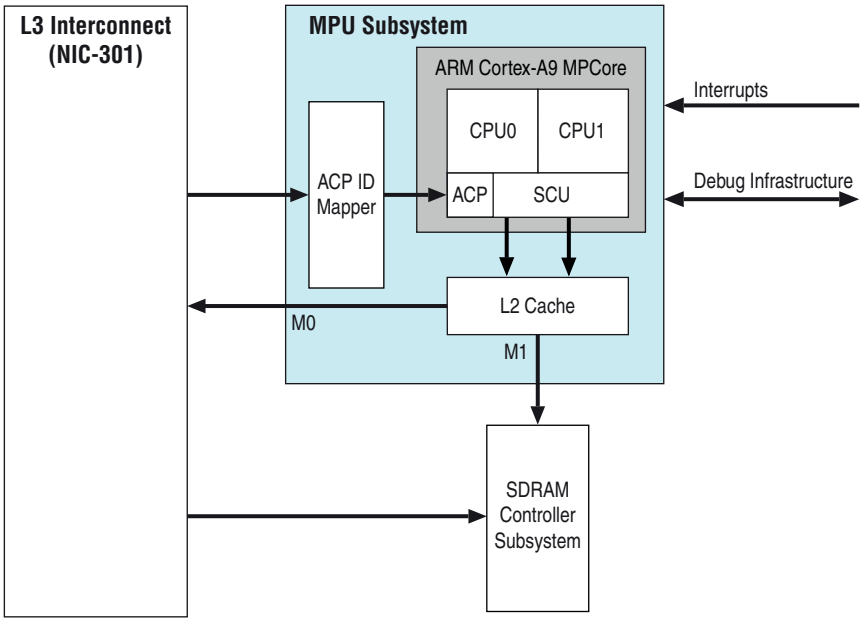
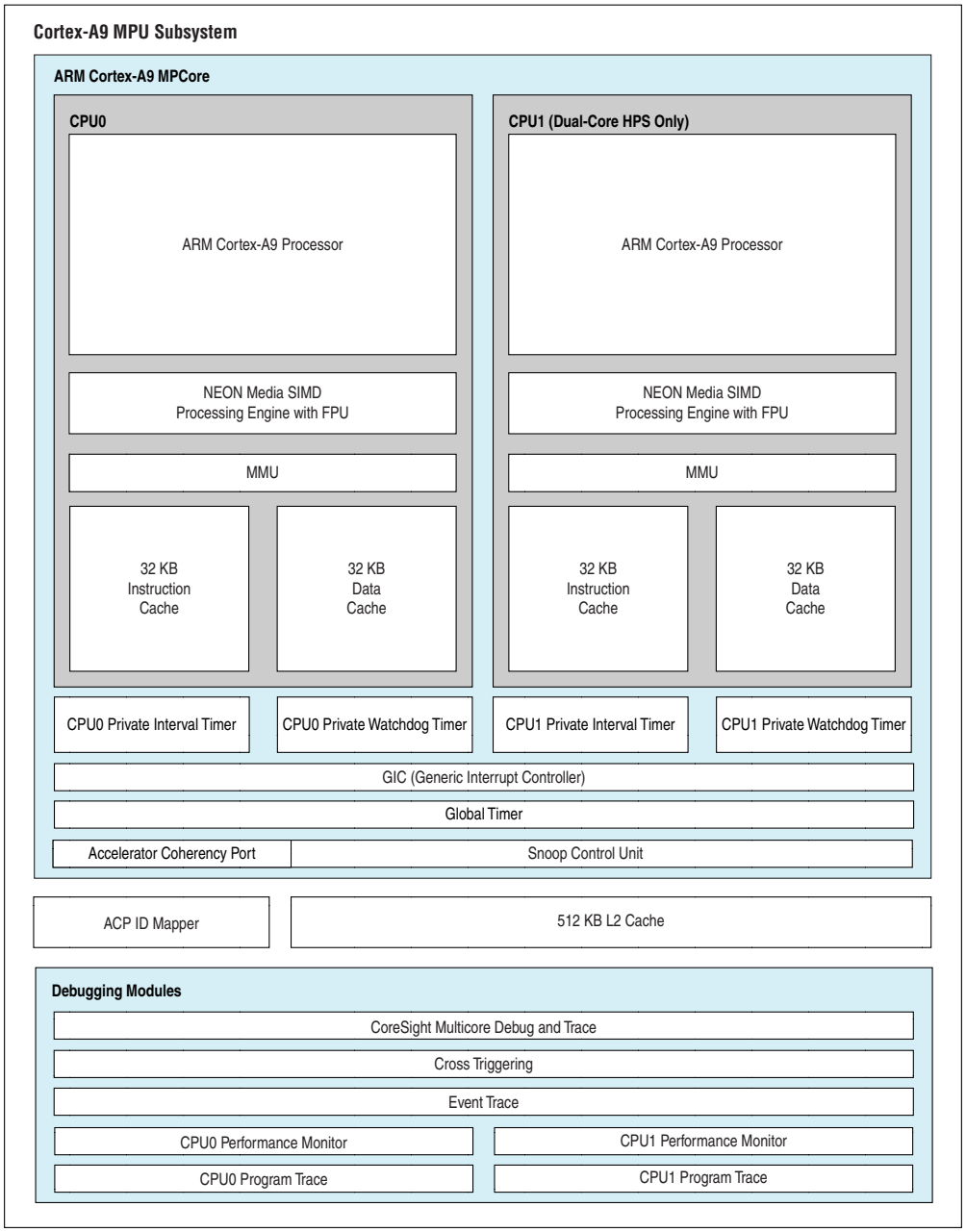


Figure 6–2 shows a block diagram of the Altera Cortex-A9 MPU subsystem.

Figure 6–2. Cortex-A9 MPU Subsystem Internals



Cortex-A9 MPU Subsystem Components

The Altera Cortex-A9 MPU subsystem consists of the following hardware blocks:

- ARM Cortex-A9 MPCore
- ARM L2C-310 L2 cache controller

- ACP ID mapper
- Debugging and trace features

This section describes the components of the Cortex-A9 MPU subsystem.

Cortex-A9 MPCore

The MPU subsystem includes a stand-alone, full-featured ARM Cortex-A9 MPCore single- or dual-core 32-bit application processor. The processor, like other HPS masters, can access IP in the FPGA fabric using through the HPS-to-FPGA bridges.

Functional Description

The ARM Cortex-A9 MPCore contains the following blocks:

- One or two Cortex-A9 Revision r3p0 processors operating in SMP or AMP mode
- Snoop control unit (SCU)
- Private interval timer for each processor core
- Private watchdog timer for each processor core
- Global timer
- Interrupt controller

Each transaction originating from the Altera Cortex-A9 MPU subsystem can be flagged as secure or nonsecure.

Implementation Details

Table 6–1 shows the parameter settings for the Altera Cortex-A9 MPCore.

Table 6–1. Cortex-A9 MPCore Processor Configuration

Feature	Options
Cortex-A9 processors	1 or 2
Instruction cache size per Cortex-A9 processor	32 KB
Data cache size per Cortex-A9 processor	32 KB
TLB size per Cortex-A9 processor	128 entries
Media Processing Engine with NEON™ technology per Cortex-A9 processor (1)	Included
Preload Engine per Cortex-A9 processor	Included
Number of entries in the Preload Engine FIFO per Cortex-A9 processor	16
Jazelle DBX extension per Cortex-A9 processor	Full
PTM interface per Cortex-A9 processor	Included
Support for parity error detection (2)	Included
ARM_BIST	Included
Master ports	Two
Accelerator Coherency Port	Included

Notes to Table 6–1:

(1) Includes support for floating-point operations.

(2) For a description of the parity error scheme and parity error signals, Refer to the Cortex-A9 Technical Reference Manual, Revision r3p0, available on the ARM website (infocenter.arm.com).

For further information about Cortex-A9 MPCore configurable options, refer to the *Introduction* chapter of the *Cortex-A9 MPCore Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

Cortex-A9 Processor

Each Cortex-A9 processor includes the following hardware blocks:

- ARM NEON™ single instruction, multiple data (SIMD) coprocessor with vector floating-point (VFP) v3 double-precision floating point unit for media and signal processing acceleration
  - Single- and double-precision IEEE-754 floating point math support
  - Integer and polynomial math support
- Level 1 (L1) cache with parity checking
  - 32 KB four-way set-associative instruction cache
  - 32 KB four-way set-associative data cache
- CoreSight™ Program Trace Macrocell (PTM) supporting instruction trace

Each Cortex-A9 processor supports the following features:

- Dual-issue superscalar pipeline with advanced branch prediction
- Out-of-order (OoO) dispatch and speculative instruction execution

- 2.5 million instructions per second (MIPS) per MHz, based on the Dhrystone 2.1 benchmark
- 128-entry translation lookaside buffer (TLB)
- TrustZone security extensions
- Configurable data endianness
- Jazelle® DBX Extensions for byte-code dynamic compiler support
- The Cortex-A9 processor architecture supports the following instruction sets:
  - The ARMv7-A performance-optimized instruction set
  - The memory-optimized Thumb®-2 mixed instruction set
    - Improves energy efficiency
    - 31% smaller memory footprint
    - 38% faster than the original Thumb instruction set
  - The Thumb instruction set—supported for legacy applications
- Each processor core in the Altera HPS includes a memory management unit (MMU) to support the memory management requirements of common modern operating systems.

The Cortex-A9 processors are designated CPU0 and CPU1.

Detailed documentation of ARM Cortex-A9 series processors, Revision r3p0, is available on the ARM website (infocenter.arm.com).

Interactive Debugging Features

Each Cortex-A9 processor has built-in debugging capabilities, including the following features:

- Six hardware breakpoints, including two with Context ID comparison capability
- Four watchpoints

The interactive debugging features can be controlled by external JTAG tools or by processor-based monitor code.

For more information about the interactive debugging system, refer to the *Debug* chapter of the *Cortex-A9 Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

L1 Caches

Cache memory that is closely coupled with an associated processor is called level 1, or L1 cache. Each Cortex-A9 processor has two independent 32 KB L1 caches—one for instructions and one for data—allowing simultaneous instruction fetches and data access.

Each L1 cache is four-way set associative, with 32 bytes per line, and supports parity checking.

Preload Engine

The preload engine (PLE) is a hardware block that enables the L2 cache to preload selected regions of memory. The PLE signals the L2 cache when a cache line will be needed in the L2 cache, by making the processor data master port start fetching the data. The processor data master does not complete the fetch or return the data to the processor. However, the L2 cache can then proceed to load the cache line. The data is only loaded to the L2 cache, not to the L1 cache or processor registers.

The preload functionality is under software control. The following PLE control parameters must be programmed:

- Programmed parameters, including the following:
  - Base address
  - Length of stride
  - Number of blocks
- A valid bit
- TrustZone memory protection for the cache memory, with an NS (non-secure) state bit
- A translation table base (TTB) address
- An Address Space Identifier (ASID) value

For more information about the PLE, refer to the *Preload Engine* chapter of the *Cortex-A9 Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

Floating Point Unit

Each ARM Cortex-A9 processor includes full support for IEEE-754 floating point operations. The floating-point unit (FPU) fully supports half-, single-, and double-precision variants of the following operations:

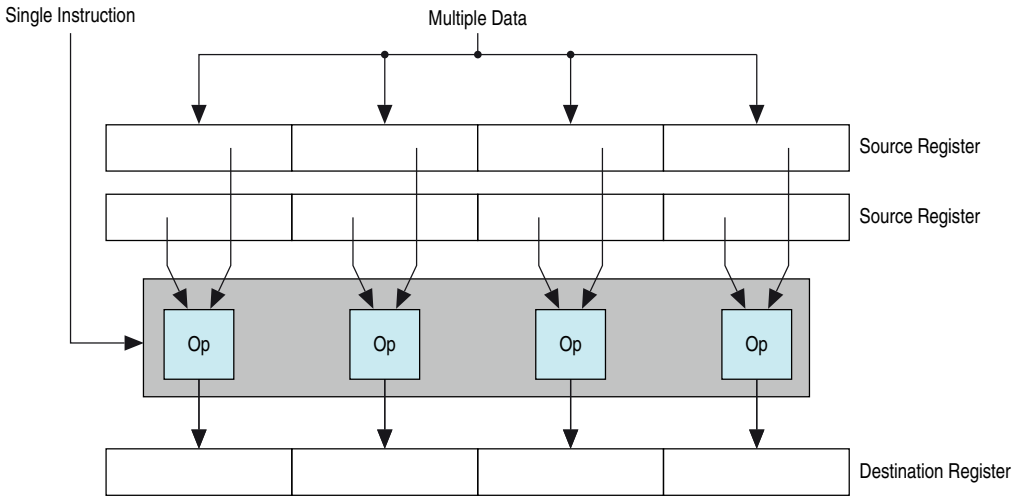
- Add
- Subtract
- Multiply
- Divide
- Multiply and accumulate (MAC)
- Square root

The FPU also converts between floating-point data formats and integers, including special operations to round towards zero required by high-level languages.

NEON Multimedia Processing Engine

The NEON multimedia processing engine (MPE) provides hardware acceleration for media and signal processing applications. Each ARM Cortex-A9 processor includes an ARM NEON MPE that supports SIMD processing, as shown in Figure 6–3. The NEON processing engine accelerates multimedia and signal processing algorithms such as video encoding and decoding, 2-D and 3-D graphics, audio and speech processing, image processing, telephony, and sound synthesis.

Figure 6–3. Single Instruction, Multiple Data (SIMD) Processing



The Cortex-A9 NEON MPE performs the following types of operations:

- SIMD and scalar single-precision floating-point computations
- Scalar double-precision floating-point computation
- SIMD and scalar half-precision floating-point conversion
- 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integer SIMD computation
- 8-bit or 16-bit polynomial computation for single-bit coefficients

The following operations are available:

- Addition and subtraction
- Multiplication with optional accumulation (MAC)
- Maximum or minimum value driven lane selection operations
- Inverse square root approximation
- Comprehensive data-structure load instructions, including register-bank-resident table lookup

For more information about the Cortex-A9 NEON MPE, refer to the *Cortex-A9 NEON™ Media Processing Engine Technical Reference Manual*, Revision r3p0, which you can download from the ARM website (infocenter.arm.com).

Memory Management Unit

The MMU is used in conjunction with the L1 and L2 caches to translate virtual addresses used by software to physical addresses used by hardware. Each processor has a private MMU.

The MMU supports the TLBs shown in Table 6–2.

Table 6–2. Supported TLBs

TLB Type	Memory Type	Number of Entries	Associativity
Micro TLB	Instruction	32	Fully associative
Micro TLB	Data	32	Fully associative
Main TLB	Instruction and Data	128	Two-way associative

The main TLB has the following features:

- Lockable entries using the lock-by-entry model
- Supports hardware page table walks to perform look-ups in the L1 data cache

For more information about the MMU, refer to the *Memory Management Unit* chapter of the *Cortex-A9 Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

The MPU address map is divided into the following regions:

- The boot region
- The SDRAM region
- The FPGA slaves region
- The HPS peripherals region

This section describes the location and contents of each region.

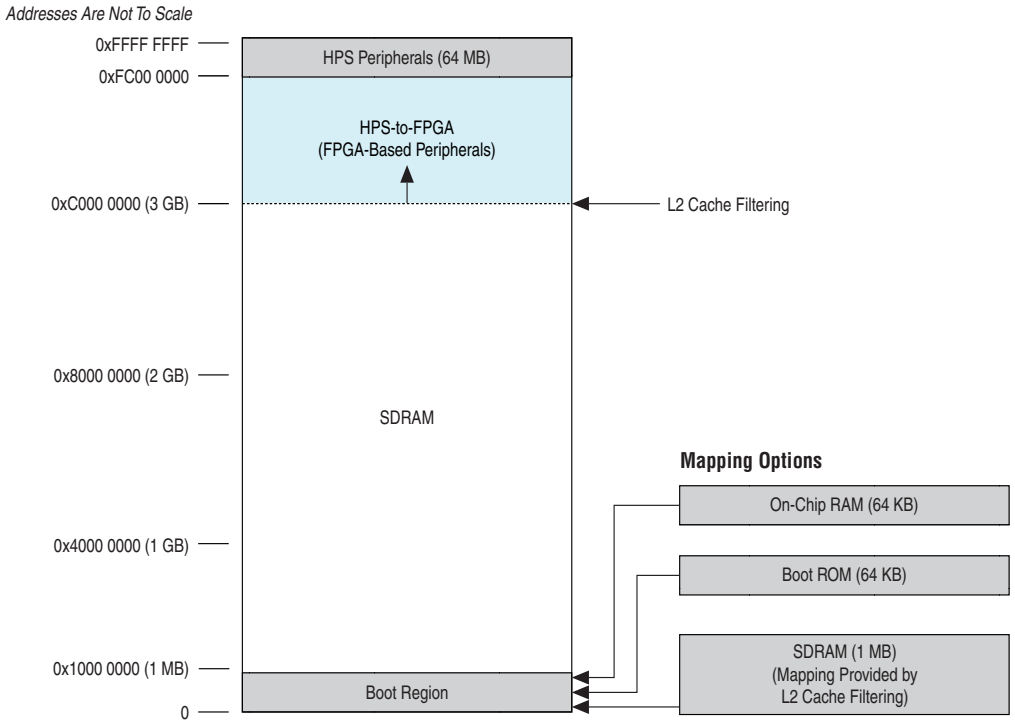
The Boot Region

The boot region is 1 MB in size, based at address 0. After power-on, or after reset of the L3 interconnect, the boot region is occupied by the boot ROM, allowing the Cortex-A9 MPCore to boot. Although the boot region size is 1 MB, accesses beyond 64 KB are illegal because the boot ROM is only 64 KB.

As shown in Figure 6–4, this 1 MB region can be subsequently remapped to the bottom 1 MB of SDRAM. For more information, refer to "The SDRAM Region".

Alternatively, the boot region can be mapped to the 64 KB on-chip RAM. For more information, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

Figure 6–4. The MPCore Address Map



The SDRAM Region

The SDRAM region starts at address `0x100000` (1 MB). The top of the region is determined by the L2 cache filter.

The L2 cache contains a filtering mechanism that routes accesses to the SDRAM and L3 interconnect. The filter defines a filter range with start and end addresses. Any access within this filter range is routed to the SDRAM subsystem. Accesses outside of this filter range are routed to the L3 interconnect.

The start and end addresses are specified in the following register fields:

- `reg12_addr_filtering_start.address_filtering_start`
- `reg12_address_filtering_end.address_filtering_end`

To remap the lower 1MB of SDRAM into the boot region, set the filter start address to `0x0` to ensure accesses between `0x0` and `0xFFFFF` are routed to the SDRAM. Independently, you can set the filter end address in 1 MB increments above `0xC0000000` to extend the upper bounds of the SDRAM region. However, you achieve this extended range at the expense of the FPGA peripheral address span. Depending on the address filter settings in the L2 cache, the top of the SDRAM region can range from `0xBFFFFFFF` to `0xFBFFFFFF`.

For more information about the L2 cache, refer to "L2 Cache" on page 6–24.

The FPGA Slaves Region

The Cortex-A9 MPU subsystem supports the variable-sized FPGA slaves region to communicate with FPGA-based peripherals. This region can start as low as `0xC0000000`, depending on the L2 cache filter settings. The top of the FPGA slaves region is located at `0xFBFFFFFF`. As a result, the size of the FPGA slaves region can range from 0 to `0x3F000000` bytes.

The HPS Peripherals Region

The HPS peripherals region is the top 64 MB in the address space, starting at `0xFC000000` and extending to `0xFFFFFFFF`. The HPS peripherals region is always allocated to the HPS dedicated peripherals for the Altera Cortex-A9 MPU subsystem.

Performance Monitoring Unit

Each Cortex-A9 processor has a Performance Monitoring Unit (PMU). The PMU supports 58 events to gather statistics on the operation of the processor and memory system. Six counters in the PMU accumulate the events in real time. The PMU counters are accessible either from the processor itself, using the Coprocessor 14 (CP14) interface, or from an external debugger. The events are also supplied to the PTM and can be used for trigger or trace.

For more information about the PMU, refer to the *Performance Monitoring Unit* chapter of the *Cortex-A9 Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

MPCore Timers

There is one interval timer and one watchdog timer for each processor.

Functional Description

Each timer is private, meaning that only its associated processor can access it. If the watchdog timer is not needed, it can be configured as a second interval timer.

Each private interval and watchdog timer has the following features:

- A 32-bit counter that optionally generates an interrupt when it reaches zero
- Configurable starting values for the counter
- An eight-bit prescaler value to qualify the clock period

Implementation Details

The timers are configurable to either single-shot or auto-reload mode. The timer blocks are clocked by mpu_periph_clk, running at ¼ the rate of mpu_clk.

For more information about private timers, refer to "About the private timer and watchdog blocks" in the *Global timer, Private timers, and Watchdog registers* chapter of the *Cortex-A9 MPCore Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

Generic Interrupt Controller

Functional Description

The Generic Interrupt Controller (GIC) supports up to 180 interrupt sources, including dedicated peripherals and IP implemented in the FPGA fabric. In a dual-core system, the GIC is shared by both Cortex-A9 processors. Each processor also has 16 banked software-generated interrupts and 16 banked private peripheral interrupts.

Implementation Details

The configuration and control for the GIC is memory-mapped and accessed through the SCU. The GIC are clocked by mpu_periph_clk, running at ¼ the rate of mpu_clk.

For more information about the GIC, refer to the *Interrupt Controller* chapter of the *Cortex-A9 MPCore Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

Table 6–3 shows the interrupt map.

Table 6–3. GIC Interrupt Map (Part 1 of 6)

GIC Interrupt Number (1)	Source Block	Interrupt Name	Combined Interrupts	Triggering
32	CortexA9_0	cpu0_parityfail	(2)	Edge
33	CortexA9_0	cpu0_parityfail_BTAC		Edge
34	CortexA9_0	cpu0_parityfail_GHB		Edge
35	CortexA9_0	cpu0_parityfail_I_Tag		Edge
36	CortexA9_0	cpu0_parityfail_I_Data		Edge
37	CortexA9_0	cpu0_parityfail_TLB		Edge
38	CortexA9_0	cpu0_parityfail_D_Outer		Edge
39	CortexA9_0	cpu0_parityfail_D_Tag		Edge
40	CortexA9_0	cpu0_parityfail_D_Data		Edge
41	CortexA9_0	cpu0_deflags0		Level
42	CortexA9_0	cpu0_deflags1		Level
43	CortexA9_0	cpu0_deflags2		Level
44	CortexA9_0	cpu0_deflags3		Level
45	CortexA9_0	cpu0_deflags4		Level
46	CortexA9_0	cpu0_deflags5		Level
47	CortexA9_0	cpu0_deflags6		Level
48	CortexA9_1	cpu1_parityfail	(3)	Edge
49	CortexA9_1	cpu1_parityfail_BTAC		Edge
50	CortexA9_1	cpu1_parityfail_GHB		Edge
51	CortexA9_1	cpu1_parityfail_I_Tag		Edge
52	CortexA9_1	cpu1_parityfail_I_Data		Edge
53	CortexA9_1	cpu1_parityfail_TLB		Edge
54	CortexA9_1	cpu1_parityfail_D_Outer		Edge
55	CortexA9_1	cpu1_parityfail_D_Tag		Edge
56	CortexA9_1	cpu1_parityfail_D_Data		Edge
57	CortexA9_1	cpu1_deflags0		Level
58	CortexA9_1	cpu1_deflags1		Level

Table 6–3. GIC Interrupt Map (Part 2 of 6)

GIC Interrupt Number (1)	Source Block	Interrupt Name	Combined Interrupts	Triggering
59	CortexA9_1	cpu1_deflags2		Level
60	CortexA9_1	cpu1_deflags3		Level
61	CortexA9_1	cpu1_deflags4		Level
62	CortexA9_1	cpu1_deflags5		Level
63	CortexA9_1	cpu1_deflags6		Level
64	SCU	scu_parityfail0	—	Edge
65	SCU	scu_parityfail1	—	Edge
66	SCU	scu_ev_abort		Edge
67	L2-Cache	l2_ecc_byte_wr_IRQ	—	Edge
68	L2-Cache	l2_ecc_corrected_IRQ	—	Edge
69	L2-Cache	l2_ecc_uncorrected_IRQ	—	Edge
70	L2-Cache	l2_combined_IRQ	(4)	Level
71	DDR	ddr_ecc_error_IRQ	—	Level
72	FPGA	FPGA_IRQ0	—	Level or Edge
73	FPGA	FPGA_IRQ1	—	Level or Edge
74	FPGA	FPGA_IRQ2	—	Level or Edge
75	FPGA	FPGA_IRQ3	—	Level or Edge
76	FPGA	FPGA_IRQ4	—	Level or Edge
77	FPGA	FPGA_IRQ5	—	Level or Edge
78	FPGA	FPGA_IRQ6	—	Level or Edge
79	FPGA	FPGA_IRQ7	—	Level or Edge
80	FPGA	FPGA_IRQ8	—	Level or Edge
81	FPGA	FPGA_IRQ9	—	Level or Edge
82	FPGA	FPGA_IRQ10	—	Level or Edge
83	FPGA	FPGA_IRQ11	—	Level or Edge
84	FPGA	FPGA_IRQ12	—	Level or Edge
85	FPGA	FPGA_IRQ13	—	Level or Edge
86	FPGA	FPGA_IRQ14	—	Level or Edge
87	FPGA	FPGA_IRQ15	—	Level or Edge
88	FPGA	FPGA_IRQ16	—	Level or Edge
89	FPGA	FPGA_IRQ17	—	Level or Edge
90	FPGA	FPGA_IRQ18	—	Level or Edge
91	FPGA	FPGA_IRQ19	—	Level or Edge
92	FPGA	FPGA_IRQ20	—	Level or Edge
93	FPGA	FPGA_IRQ21	—	Level or Edge
94	FPGA	FPGA_IRQ22	—	Level or Edge
95	FPGA	FPGA_IRQ23	—	Level or Edge
96	FPGA	FPGA_IRQ24	—	Level or Edge
97	FPGA	FPGA_IRQ25	—	Level or Edge

Table 6–3. GIC Interrupt Map (Part 3 of 6)

GIC Interrupt Number (1)	Source Block	Interrupt Name	Combined Interrupts	Triggering
98	FPGA	FPGA_IRQ26	—	Level or Edge
99	FPGA	FPGA_IRQ27	—	Level or Edge
100	FPGA	FPGA_IRQ28	—	Level or Edge
101	FPGA	FPGA_IRQ29	—	Level or Edge
102	FPGA	FPGA_IRQ30	—	Level or Edge
103	FPGA	FPGA_IRQ31	—	Level or Edge
104	FPGA	FPGA_IRQ32	—	Level or Edge
105	FPGA	FPGA_IRQ33	—	Level or Edge
106	FPGA	FPGA_IRQ34	—	Level or Edge
107	FPGA	FPGA_IRQ35	—	Level or Edge
108	FPGA	FPGA_IRQ36	—	Level or Edge
109	FPGA	FPGA_IRQ37	—	Level or Edge
110	FPGA	FPGA_IRQ38	—	Level or Edge
111	FPGA	FPGA_IRQ39	—	Level or Edge
112	FPGA	FPGA_IRQ40	—	Level or Edge
113	FPGA	FPGA_IRQ41	—	Level or Edge
114	FPGA	FPGA_IRQ42	—	Level or Edge
115	FPGA	FPGA_IRQ43	—	Level or Edge
116	FPGA	FPGA_IRQ44	—	Level or Edge
117	FPGA	FPGA_IRQ45	—	Level or Edge
118	FPGA	FPGA_IRQ46	—	Level or Edge
119	FPGA	FPGA_IRQ47	—	Level or Edge
120	FPGA	FPGA_IRQ48	—	Level or Edge
121	FPGA	FPGA_IRQ49	—	Level or Edge
122	FPGA	FPGA_IRQ50	—	Level or Edge
123	FPGA	FPGA_IRQ51	—	Level or Edge
124	FPGA	FPGA_IRQ52	—	Level or Edge
125	FPGA	FPGA_IRQ53	—	Level or Edge
126	FPGA	FPGA_IRQ54	—	Level or Edge
127	FPGA	FPGA_IRQ55	—	Level or Edge
128	FPGA	FPGA_IRQ56	—	Level or Edge
129	FPGA	FPGA_IRQ57	—	Level or Edge
130	FPGA	FPGA_IRQ58	—	Level or Edge
131	FPGA	FPGA_IRQ59	—	Level or Edge
132	FPGA	FPGA_IRQ60	—	Level or Edge
133	FPGA	FPGA_IRQ61	—	Level or Edge
134	FPGA	FPGA_IRQ62	—	Level or Edge
135	FPGA	FPGA_IRQ63	—	Level or Edge

Table 6–3. GIC Interrupt Map (Part 4 of 6)

GIC Interrupt Number (1)	Source Block	Interrupt Name	Combined Interrupts	Triggering
136	DMA	dma_IRQ0	—	Level
137	DMA	dma_IRQ1	—	Level
138	DMA	dma_IRQ2	—	Level
139	DMA	dma_IRQ3	—	Level
140	DMA	dma_IRQ4	—	Level
141	DMA	dma_IRQ5	—	Level
142	DMA	dma_IRQ6	—	Level
143	DMA	dma_IRQ7	—	Level
144	DMA	dma_irq_abort	—	Level
145	DMA	dma_ecc_corrected_IRQ		Level
146	DMA	dma_ecc_uncorrected_IRQ		Level
147	EMAC0	emac0_IRQ	(5)	Level
148	EMAC0	emac0_tx_ecc_corrected_IRQ		Level
149	EMAC0	emac0_tx_ecc_uncorrected_IRQ		Level
150	EMAC0	emac0_rx_ecc_corrected_IRQ		Level
151	EMAC0	emac0_rx_ecc_uncorrected_IRQ		Level
152	EMAC1	emac1_IRQ	(5)	Level
153	EMAC1	emac1_tx_ecc_corrected_IRQ		Level
154	EMAC1	emac1_tx_ecc_uncorrected_IRQ		Level
155	EMAC1	emac1_rx_ecc_corrected_IRQ		Level
156	EMAC1	emac1_rx_ecc_uncorrected_IRQ		Level
157	USB0	usb0_IRQ		Level
158	USB0	usb0_ecc_corrected_IRQ		Level
159	USB0	usb0_ecc_uncorrected_IRQ		Level
160	USB1	usb1_IRQ		Level
161	USB1	usb1_ecc_corrected_IRQ		Level
162	USB1	usb1_ecc_uncorrected_IRQ		Level
163	CAN0	can0_sts_IRQ		Level
164	CAN0	can0_mo_IRQ		Level
165	CAN0	can0_ecc_corrected_IRQ		Level
166	CAN0	can0_ecc_uncorrected_IRQ		Level
167	CAN1	can1_sts_IRQ		Level
168	CAN1	can1_mo_IRQ		Level
169	CAN1	can1_ecc_corrected_IRQ		Level
170	CAN1	can1_ecc_uncorrected_IRQ		Level
171	SDMMC	sdmmc_IRQ		Level
172	SDMMC	sdmmc_porta_ecc_corrected_IRQ		Level
173	SDMMC	sdmmc_porta_ecc_uncorrected_IRQ		Level

Table 6–3. GIC Interrupt Map (Part 5 of 6)

GIC Interrupt Number (1)	Source Block	Interrupt Name	Combined Interrupts	Triggering
174	SDMMC	sdmmc_portb_ecc_corrected_IRQ		Level
175	SDMMC	sdmmc_portb_ecc_uncorrected_IRQ		Level
176	NAND	nand_IRQ		Level
177	NAND	nandr_ecc_corrected_IRQ		Level
178	NAND	nandr_ecc_uncorrected_IRQ		Level
179	NAND	nandw_ecc_corrected_IRQ		Level
180	NAND	nandw_ecc_uncorrected_IRQ		Level
181	NAND	nande_ecc_corrected_IRQ		Level
182	NAND	nande_ecc_uncorrected_IRQ		Level
183	QSPI	qspi_IRQ		Level
184	QSPI	qspi_ecc_corrected_IRQ		Level
185	QSPI	qspi_ecc_uncorrected_IRQ		Level
186	SPI0	spi0_IRQ	(6)	Level
187	SPI1	spi1_IRQ	(6)	Level
188	SPI2	spi2_IRQ	(6)	Level
189	SPI3	spi3_IRQ	(6)	Level
190	I2C0	i2c0_IRQ	(7)	Level
191	I2C1	i2c1_IRQ	(7)	Level
192	I2C2	i2c2_IRQ	(7)	Level
193	I2C3	i2c3_IRQ	(7)	Level
194	UART0	uart0_IRQ		Level
195	UART1	uart1_IRQ		Level
196	GPIO0	gpio0_IRQ	—	Level
197	GPIO1	gpio1_IRQ	—	Level
198	GPIO2	gpio2_IRQ	—	Level
199	Timer0	timer_l4sp_0_IRQ	(8)	Level
200	Timer1	timer_l4sp_1_IRQ	(8)	Level
201	Timer2	timer_osc1_0_IRQ	(8)	Level
202	Timer3	timer_osc1_1_IRQ	(8)	Level
203	Watchdog0	wdog0_IRQ	—	Level
204	Watchdog1	wdog1_IRQ	—	Level
205	Clock manager	clkmgr_IRQ		Level
206	Clock manager	mpuwakeup_IRQ		Level
207	FPGA manager	fpga_man_IRQ	(9)	Level
208	CoreSight	nCTIIRQ[0]		Level
209	CoreSight	nCTIIRQ[1]		Level

Table 6–3. GIC Interrupt Map (Part 6 of 6)

GIC Interrupt Number (1)	Source Block	Interrupt Name	Combined Interrupts	Triggering
210	On-chip RAM	ram_ecc_corrected_IRQ		Level
211	On-chip RAM	ram_ecc_uncorrected_IRQ		Level

Notes to Table 6–3:

(1) To ensure that you are using the correct GIC interrupt number, your code should refer to the symbolic interrupt name, as shown in the **Interrupt Name** column. Symbolic interrupt names are defined in a header file distributed with the source installation for your operating system.

(2) This interrupt combines the interrupts named cpu0_parityfail_*.

(3) This interrupt combines the interrupts named cpu1_parityfail_*.

(4) This interrupt combines the following interrupts: DECERRINTR, ECNTRINTR, ERRRDINTR, ERRRTINTR, ERRWDINTR, ERRWTINTR, PARRDINTR, PARRTINTR, and SLVERRINTR.

(5) This interrupt combines sbd_intr_o, lpi_intr_o, and pmt_intr_o.

(6) This interrupt combines the following interrupts: ssi_txe_intr, ssi_txo_intr, ssi_rxf_intr, ssi_rxo_intr, ssi_rxu_intr, and ssi_mst_intr.

(7) This interrupt combines the following interrupts: ic_rx_under_intr, ic_rx_full_intr, ic_tx_over_intr, ic_tx_empty_intr, ic_rd_req_intr, ic_tx_abrt_intr, ic_rx_done_intr, ic_activity_intr, ic_stop_det_intr, ic_start_det_intr, and ic_gen_call_intr.

(8) This interrupt combines TIMINT1 and TIMINT2.

(9) This interrupt combines the following interrupts: fpga_man_irq[7..0].

Global Timer

The MPU features a global 64-bit, auto-incrementing timer, which is primarily used by the operating system.

Functional Description

The global timer is accessible by the processors using memory-mapped access through the SCU. The global timer has the following features:

- 64-bit incrementing counter with an auto-incrementing feature. It continues incrementing after sending interrupts.
- Memory-mapped in the private memory region.
- Accessed at reset in Secure State only. It can only be set once, but secure code can read it at any time.
- Accessible to both Cortex-A9 processors in the MPCore.

Implementation Details

Each Cortex-A9 processor has a private 64-bit comparator that generates a private interrupt when the counter reaches the specified value. Each Cortex-A9 processor uses the banked ID, ID27, for this interrupt. ID27 is sent to the GIC as a Private Peripheral Interrupt (PPI).

The global timer are clocked by mpu_periph_clk, running at ¼ the rate of mpu_clk.

For more information about the global timer, refer to "About the Global Timer" in the *Global timer, Private timers, and Watchdog registers* chapter of the *Cortex-A9 MPCore Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

Snoop Control Unit

The SCU manages data traffic for the Cortex-A9 processors and the memory system, including the L2 cache. In a multi-master system, the processors and other masters can operate on shared data. The SCU ensures that each processor operates on the most up-to-date copy of data, maintaining cache coherency.

Functional Description

The SCU is used to connect the Cortex-A9 processors and the ACP to the L2 cache controller. The SCU performs the following functions:

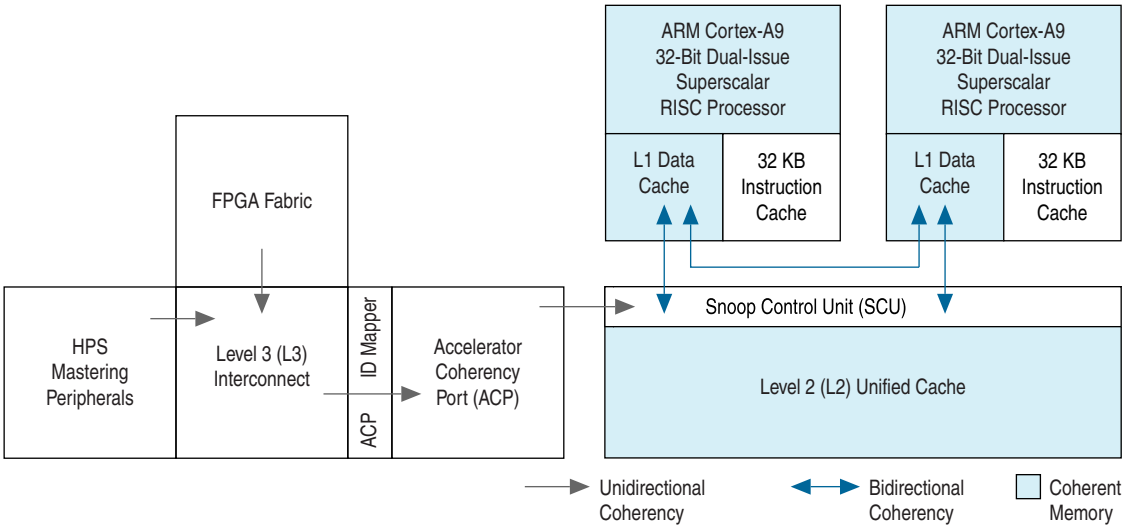
- When the processors are set to SMP mode, the SCU maintains data cache coherency between the processors.

  The SCU does not maintain coherency of the instruction caches.

- Initiates L2 cache memory accesses
- Arbitrates between processors requesting L2 access
- Manages ACP access with cache coherency capabilities.

Figure 6–5 shows the SCU in a dual-processor system, illustrating the flow of data among the L1 data caches and the SCU.

Figure 6–5. Coherent Memory, Snoop Control Unit, and Accelerator Coherency Port



For more information about the SCU, refer to the *Snoop Control Unit* chapter of the *Cortex-A9 MPCore Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

Implementation Details

When the processor writes to any coherent memory location, the SCU ensures that the relevant data is coherent (updated, tagged, or invalidated). Similarly, the SCU monitors read operations from a coherent memory location. If the required data is already stored within the other processor's L1 cache, the data is returned directly to the requesting processor. If the data is not in L1 cache, the SCU issues a read to the L2 cache. If the data is not in the L2 cache memory, the read is finally forwarded to main memory. The primary goal is to minimize power consumption and maximize overall memory performance.

The SCU maintains bidirectional coherency between the L1 data caches belonging to the processors. When one processor writes to a location in its L1 cache, if the same location is cached in the other L1 cache, the SCU updates it.

Non-coherent data passes through as a standard read or write operation.

The SCU also arbitrates between the Cortex-A9 processors if both attempt simultaneous access to the L2 cache, and manages accesses from the ACP.

Accelerator Coherency Port

The ACP allows peripherals—including FPGA-based peripherals—to maintain data coherency with the Cortex-A9 MPCore processors and the SCU. As shown in Figure 6–5 on page 6–18, dedicated peripherals in the HPS, and those built in FPGA logic, access the coherent memory through the ACP ID mapper and the ACP. For information about the ACP ID mapper, refer to "ACP ID Mapper" on page 6–20.

The high-bandwidth peripherals, including the FPGA data ports, connect to the L3 interconnect.

Burst Sizes and Byte Strobes

The ACP improves system performance for hardware accelerators in the FPGA fabric. However, in order to achieve high levels of performance, you must use the burst types listed in Table 6–4. The other burst types have significantly lower performance.

Table 6–4. Recommended Burst Types

Burst Type	Beats	Width (Bits)	Address Type	Byte Strobes
Wrapping	4	64	64-bit aligned	Asserted
Incrementing	4	64	32-bit aligned	Asserted

If the slave port of the FPGA-to-HPS bridge is not 64 bits wide, you must supply bursts to the FPGA-to-HPS bridge that are upsized or downsized to the burst types above. For example, if the slave data width of the FPGA-to-HPS bridge is 32 bits, then bursts of eight beats by 32 bits are required to access the ACP efficiently.

If the address and burst size of the transaction to the ACP matches either of the conditions above, the logic in the MPU assumes the transaction has all its byte strobes set. If the byte strobes are not all set, then the write does not actually overwrite all the bytes in the word. Instead, the cache assumes the whole cache line is valid. If this line is dirty (and therefore gets written out to SDRAM), data corruption might occur.

Exclusive and Locked Accesses

The ACP does not support exclusive accesses to coherent memory. The ACP supports exclusive accesses to non-coherent memory; however, it is important that the exclusive access transaction is not affected by the upsizing and downsizing logic of the FPGA-to-HPS bridge or the L3 interconnect. If the exclusive access is broken into multiple transactions due to the sizing logic, the exclusive access bit is cleared by the bridge or interconnect and the exclusive access fails.

Altera recommends that exclusive accesses bypass the ACP altogether, either through the 32-bit slave port of the SDRAM controller connected directly to the L3 interconnect or through the FPGA-to-SDRAM interface.

For more information about the exclusive access support of the SDRAM controller subsystem, refer to the *SDRAM Controller Subsystem* chapter in volume 3 of the *Cyclone® V Device Handbook*.

The ACP ID mapper does not support locked accesses. To ensure mutually exclusive access to shared data, use the exclusive access support built into the SDRAM controller.

ACP ID Mapper

The ACP ID mapper is situated between the level 3 (L3) interconnect and the MPU subsystem ACP slave. It is responsible for mapping 12-bit Advanced Microcontroller Bus Architecture (AMBA®) Advanced eXtensible Interface (AXI™) IDs (input IDs) from the L3 interconnect to 3-bit AXI IDs (output IDs) supported by the ACP slave port.

The ACP ID mapper also implements a 1 GB coherent window into 4 GB address space.

Functional Description

The ACP slave supports up to six masters. However, custom peripherals implemented in the FPGA fabric can have a larger number of masters that need to access the ACP slave. The ACP ID mapper allows these masters to access the ACP.

The ACP ID mapper resides between the interconnect and the ACP slave of the MPU subsystem. It has the following characteristics:

- Support for up to six concurrent ID mappings
- 1 GB coherent window into 4 GB MPCore address space
- Remaps the 5-bit user sideband signals used by the Snoop Control Unit (SCU) and L2 cache.

For more information about AXI user sideband signals, refer to the *CoreLink Level 2 Cache Controller L2C-310 Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

Implementation Details

The ACP is accessed by masters that require access to coherent memory. The ACP slave port can be accessed by the master peripherals of the L3 interconnect, as well as by masters implemented in the FPGA fabric (via the FPGA-to-HPS bridge). Figure 6–1 on page 6–2 shows the ACP ID mapper.

The ACP ID mapper supports the following ID mapping modes:

- Dynamic mapping
- Fixed mapping

Software can select the ID mapping on a per-ID basis. For input IDs that are configured for fixed mapping, there is a one-to-one mapping from input IDs to output IDs. When an input ID is configured for dynamic mapping, it is automatically mapped to an available output ID. The dynamic mode is more flexible because the hardware handles the mapping. The hardware mapping allows you to use one output ID for more than one input ID. Output IDs are assigned to input IDs on a first-come, first-served basis.

Out of the total of eight output IDs, only six are available to masters of the L3 interconnect. The first two output IDs (0 and 1) are dedicated to the Cortex-A9 processor cores in the MPU subsystem, leaving the last six output IDs (2-7) available to the ACP ID mapper. Output IDs 2-6 support fixed and dynamic modes of operation while output ID 7 supports dynamic only.

The operating modes are programmable through accesses to the control and status registers in the ACP ID mapper, available via the level 4 peripheral bus connection. At reset time, the ACP ID mapper defaults to dynamic ID mapping for all output IDs except ID 2, which resets to a fixed mapping for the Debug Access Port (DAP) input ID.

Table 6–1 summarizes the expected usage of the 3-bit output IDs, and their settings at reset.

Table 6–5. ID Intended Usage

Output ID	Reset State	Intended Use
7	Dynamic	Dynamic mapping only
6	Dynamic	Fixed or dynamic, programmed by software.
5		
4		
3		
2	Fixed at 0x001 (DAP)	Assigned to the input ID of the DAP at reset. After reset, can be either fixed or dynamic, programmed by software.
1	—	Not used by the ACP ID Mapper
0		

For masters that cannot drive the AXI user sideband signal of incoming transactions, the ACP ID mapper can control overriding this signal. The ACP ID mapper can also control which 1 GB coherent window into memory is accessed by masters of the L3 interconnect. Each fixed mapping can be assigned a different user sideband signal and memory window to allow specific settings for different masters. All dynamic mappings share a common user sideband signal and memory window setting.

Transaction Capabilities

At any one time, the ACP ID mapper can accept and issue up to 15 transactions per ID mapping. Read and write ID mappings are managed in separate lists, allowing more unique input IDs to be remapped at any given time. If a master issues a series of reads and writes with the same input ID, there are no ordering restrictions.

Because there are only six output IDs available, there can be no more than six read and six write transactions with unique IDs in progress at any one time. The write acceptance of the ACP slave is five transactions, and the read acceptance is 13 transactions. Only four coherent read transactions per ID mapping can be outstanding at one time.

Dynamic Mapping Mode

In dynamic mode, every unique input ID that is received from the L3 master port is assigned to an unused output ID. The new output ID is applied to the transaction as it is issued to the ACP slave of the SCU. Any transaction that arrives to the ACP ID mapper with an input ID that matches an already-in-progress transaction is mapped to the same output ID. Once all transactions on an ID mapping have completed, that output ID is released and can be used again for other input IDs.

Fixed Mapping Mode

In fixed mode, output IDs 2 through 6 can be assigned by software to a specific 12-bit input ID. This ability makes it possible to use the lock-by-master feature of the L2 cache controller, because the input transaction ID from the master is always assigned to a specific output ID. Unlike dynamic mode, ID 7 is not available for fixed mapping because it is reserved for dynamic mode only to avoid system deadlocks.

The ACP ID mapper has two banks of registers to control the behavior of the mappings, namely, a request bank and a read-only status bank. Both banks contain the same number of registers. To change the settings for a particular mapping (either a specific fixed ID, or all dynamic mappings), software should write to the appropriate register in the request bank. The hardware examines the request, and only applies the change when safe to do so, which is when there are no outstanding transactions with the output ID. When the change is applied, the status register is updated. Software should check that the change has actually taken place by polling the corresponding status register.

Table 6–6 shows the input IDs issued from the interconnect for each HPS peripheral master that can access the ACP ID mapper.

Table 6–6. HPS Peripheral Master Input IDs

Interconnect Master	ID (1)
DMA	00000xxxx011
EMAC0	10000xxxx001
EMAC1	10000xxxx010
USB0	100000000011
USB1	100000000110
NAND	1xxxxxxxx100
ETR	100000000000
DAP	000000000001
SD/MMC	100000000101
FPGA-to-HPS bridge	0xxxxxxxx100

Notes to Table 6–6:

(1) Values are in binary. The letter *x* denotes variable ID bits each master passes with each transaction.

Control of the AXI User Sideband Signals

The ACP ID mapper module allows control of the AXI user sideband signal values. Not all masters drive these signals, so the ACP ID mapper makes it possible to drive the 5-bit user sideband signal with either a default value (in dynamic mode) or specific values (in fixed mode).

There are registers available to configure the default values of the user sideband signals for all transactions, and fixed values of these signals for particular transactions in fixed mapping mode. In dynamic mode, the user sideband signals of incoming transactions are mapped with the default values stored in the register. In fixed mapping mode, the input ID of the transaction is mapped to the 3-bit output ID and the user sideband signals of the transaction are mapped with the values stored in the register that corresponds to the output ID. One important exception, however, is that the ACP ID mapper always allows user sideband signals from the FPGA-to-HPS bridge to pass through to the ACP regardless of the user sideband value associated with the ID.

Memory Region Remap

The ACP ID mapper has 1 GB of address space, which is by default a view into the bottom 1 GB of SDRAM. The mapper also allows transactions to be routed to different 1 GB-sized memory regions, called pages, in both dynamic and fixed modes. The two most significant bits of incoming 32-bit AXI address signals are replaced with the 2-bit user-configured address page decode information. The page decoder uses the values shown in Table 6–7.

Table 6–7. Page Decoder Values

Page	Address Range
0	0x00000000–0x3FFFFFFF
1	0x40000000–0x7FFFFFFF
2	0x80000000–0xBFFFFFFF
3	0xC0000000–0xFFFFFFFF

With this page decode information, a master can read or write to any 1 GB region of the 4 GB memory space while maintaining cache coherency with the MPU subsystem.

Using this feature, a debugger can have a coherent view into main memory, without having to stop the processor. For example, at reset the DAP input ID (0x001) is mapped to output ID 2, so the debugger can vary the 1 GB window that the DAP accesses without affecting any other traffic flow to the ACP.

L2 Cache

The MPU subsystem includes a secondary 512 KB L2 shared, unified cache memory.

Functional Description

The L2 cache is much larger than the L1 cache. The L2 cache has significantly lower latency than external memory. The L2 cache is up to eight-way associative, configurable down to one-way (direct mapped). Like the L1 cache, the L2 cache can be locked by cache line, locked by way, or locked by bus master.

The L2 cache implements error correction codes (ECCs) and ECC error reporting. The cache can report a number of events to the processor and operating system.

The L2 cache consists of the ARM L2C-310 L2 cache controller configured as follows:

- 512 KB total memory
- Eight-way associativity
- Physically addressed, physically tagged
- Line length of 32 bytes

- Critical first word linefills
- Support for all AXI cache modes, as shown in Table 6–8.

Table 6–8. AXI Cache Mode Support

Cache Mode
Write-through (1)
Write-back (1)
Read allocate
Write allocate
Read and write allocate

Note to Table 6–8:

(1) Restrictions exist when using ECCs. For more information about SEU protection, refer to the *System Manager* chapter in volume 3of the *Cyclone V Device Handbook*.

- Single event upset (SEU) protection
  - Parity on Tag RAM
  - ECC on L2 Data RAM

  For more information about SEU protection, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

- Two slave ports mastered by the SCU
- Two master ports connected to the following slave ports:
  - SDRAM controller, 64 bit slave port width
  - L3 interconnect, 64 bit slave port width
- Cache lockdown capabilities as follows:
  - Line lockdown
  - Lockdown by way
  - Lockdown by master (both processors and ACP masters)
- TrustZone support
- Cache event monitoring. For more information, refer to "L2 Cache Event Monitoring" on page 6–27.

Figure 6–1 on page 6–2 shows the L2 cache. The L2 cache can access either the L3 interconnect fabric or the SDRAM. The L2 cache address filtering determines how much address space is allocated to the HPS-to-FPGA bridge and how much is allocated to SDRAM, as described in "Memory Management Unit" on page 6–9.

ECC Support

The L2 cache has the option of using ECCs to protect against SEU errors in the cache RAM.

Enabling ECCs does not affect the performance of the L2 cache. The ECC bits are calculated only for writes to the data RAM that are 64 bits wide (8 bytes, or one-quarter of the cache line length). The ECC logic does not perform a read-modify-write when calculating the ECC bits. The ECC protection bits are not valid in the following cases:

- Data is written that is not 64-bit aligned in memory
- Data is written that is less than 64 bits in width

In these cases the Byte Write Error interrupt is asserted. Cache data is still written when such an error occurs. However, the ECC error detection and correction continues to function. Therefore, the cache data is likely to be incorrect on subsequent reads.

To use ECCs, the software and system must meet the following requirements:

- L1 and L2 cache must be configured as write-back allocate for any cacheable memory region
- FPGA soft IP using the ACP must only perform the following types of data writes:
  - 64-bit aligned in memory
  - 64 bit wide accesses

For more information about SEU errors, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Implementation Details

Table 6–9 shows the parameter settings for the cache controller.

Table 6–9. Cache Controller Configuration

Feature	Meaning
Cache way size	64 KB
Number of cache ways	8 ways
RAM latencies	2 cycles of latency
Parity logic	Parity logic enabled
Lockdown by master	Lockdown by master enabled
Lockdown by line	Lockdown by line enabled
AXI ID width on slave ports	6 AXI ID bits on slave ports
Address filtering	Address filtering logic enabled
Speculative read	Logic for supporting speculative read enabled
Presence of ARUSERMx and AWUSERMx sideband signals	Sideband signals enabled

For further information about cache controller configurable options, refer to the *CoreLink Level 2 Cache Controller L2C-310 Technical Reference Manual*, Revision r3p2, available on the ARM website (infocenter.arm.com).

L2 Cache Lockdown Capabilities

The L2 cache has three methods to lock data in the cache RAMs:

- Lockdown by line—Used to lock lines in the cache. This is commonly used for loading critical sections of software into the cache temporarily.
- Lockdown by way—Allows any or all of the eight cache ways to be locked. This is commonly used for loading critical data or code into the cache.
- Lockdown by master—Allows cache ways to be dedicated to a single master port. This allows a large cache to look like smaller caches to multiple master ports. The L2 cache can be mastered by CPU0, CPU1, or the six ACP masters, for a total of eight possible master ports.

For more information about L2 cache lockdown capabilities, refer to "Cache operation" in the *Functional Overview* chapter of the *CoreLink Level 2 Cache Controller L2C-310 Technical Reference Manual*, Revision r3p2, available on the ARM website (infocenter.arm.com).

L2 Cache Event Monitoring

The L2 cache supports the built-in cache event monitoring signals shown in Table 6–10. The L2 cache can count two of the events at any one time.

Table 6–10. L2 Cache Events

Event	Description
CO	Eviction (cast out) of a line from the L2 cache.
DRHIT	Data read hit in the L2 cache.
DRREQ	Data read lookup to the L2 cache. Subsequently results in a hit or miss.
DWHIT	Data write hit in the L2 cache.
DWREQ	Data write lookup to the L2 cache. Subsequently results in a hit or miss.
DWTREQ	Data write lookup to the L2 cache with write-through attribute. Subsequently results in a hit or miss.
EPFALLOC	Prefetch hint allocated into the L2 cache.
EPFHIT	Prefetch hint hits in the L2 cache.
EPFRCVDS0	Prefetch hint received by slave port S0.
EPFRCVDS1	Prefetch hint received by slave port S1.
IPFALLOC	Allocation of a prefetch generated by L2 cache controller into the L2 cache.
IRHIT	Instruction read hit in the L2 cache.
IRREQ	Instruction read lookup to the L2 cache. Subsequently results in a hit or miss.
SPNIDEN	Secure privileged non-invasive debug enable.
SRCONFS0	Speculative read confirmed in slave port S0.
SRCONFS1	Speculative read confirmed in slave port S1.
SRRCVDS0	Speculative read received by slave port S0.
SRRCVDS1	Speculative read received by slave port S1.
WA	Allocation into the L2 cache caused by a write, with write-allocate attribute, miss.

For more information about the built-in L2 event monitoring capability, refer to "Implementation details" in the *Functional Overview* chapter of the *CoreLink Level 2 Cache Controller L2C-310 Technical Reference Manual*, Revision r3p2, available on the ARM website (infocenter.arm.com).

In addition, the L2 cache events can be captured and timestamped using dedicated debugging circuitry.

For more information about L2 event capture, refer to the *Debug* chapter of the *Cortex-A9 MPCore Technical Reference Manual*, Revision r3p0, available on the ARM website (infocenter.arm.com).

Debugging Modules

The MPU subsystem includes debugging resources through ARM CoreSight on-chip debugging and trace. The following functionality is included:

- Individual program trace for each processor
- Event trace for the Cortex-A9 MPCore
- Cross triggering between processors and other HPS debugging features

Program Trace

Each processor has an independent PTM that provides real-time instruction flow trace. The PTM is compatible with a number of third-party debugging tools.

The PTM provides trace data in a highly compressed format. The trace data includes tags for specific points in the program execution flow, called waypoints. Waypoints are specific events or changes in the program flow.

The PTM recognizes and tags the waypoints listed in Table 6–11.

Table 6–11. Waypoints Supported by the PTM

Type	Additional Waypoint Information
Indirect branches	Target address and condition code
Direct branches	Condition code
Instruction barrier instructions	—
Exceptions	Location where the exception occurred
Changes in processor instruction set state	—
Changes in processor security state	—
Context ID changes	—
Entry to and return from debug state when Halting debug mode is enabled	—

The PTM optionally provides additional information for waypoints, including the following.

- Processor cycle count between waypoints
- Global timestamp values

  For information about global timestamps, refer to the *CoreSight Debug and Trace* chapter in volume 3 of the *Cyclone V Device Handbook.*

- Target addresses for direct branches

For more information about the PTM, refer to the *CoreSight PTM-A9 Technical Reference Manual*, Revision r1p0, available on the ARM website (infocenter.arm.com).

Event Trace

Events from each processor can be used as inputs to the PTM. The PTM can use these events as trace and trigger conditions.

For more information about the event trace, refer to "Performance Monitoring Unit" on page 6–11.

For more information about the trigger and trace capabilities, refer to the *CoreSight PTM-A9 Technical Reference Manual*, Revision r1p0, available on the ARM website (infocenter.arm.com).

Cross-Triggering

The PTM can export trigger events and perform actions on trigger inputs. The cross-trigger signals interface with other HPS debugging components including the FPGA fabric. Also, a breakpoint in one processor can trigger a break in the other.

For detailed information about cross-triggering, refer to the *CoreSight Debug and Trace* chapter in volume 3 of the *Cyclone V Device Handbook.*

For more information about debugging hardware in the MPU, refer to the *CoreSight Debug and Trace* chapter in volume 3 of the *Cyclone V Device Handbook.*

Cortex-A9 MPU Subsystem Register Implementation

The following configurations are available through registers in the Cortex-A9 subsystem:

- All processor-related controls, including the MMU and L1 caches, are controlled using the Coprocessor 15 (CP15) registers of each individual processor.
- All SCU registers, including control for the timers and GIC, are memory map accessible
- All L2 cache registers are memory-mapped.

For an address map of peripheral slave ports, including the SCU and L2 cache, refer to the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*. For detailed definitions of the registers for the Altera Cortex-A9 MPU subsystem, refer to the *Cortex-A9 MPCore Technical Reference Manual*, Revision r3p0, and the *CoreLink Level 2 Cache Controller L2C-310 Technical Reference Manual*, Revision r3p2, available on the ARM website (infocenter.arm.com).

Document Revision History

Table 6–12 shows the revision history for this document.

Table 6–12. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	■ Add description of the ACP ID mapper ■ Consolidate redundant information
January 2012	1.0	Initial release.

Section IV. Debug and Trace

cv_54001-1.0

This section includes the following chapters:

- Chapter 7, CoreSight Debug and Trace

For information about the revision history for chapters in this section, refer to "Document Revision History" in each individual chapter.

7. CoreSight Debug and Trace

cv_54007-1.2

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.

The HPS contains the following ARM CoreSight debug components:

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 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
- Instruction trace interface through TPIU for trace analysis tools
- Custom message injection through STM into trace stream for delivery to 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 ETR AXI master connected to the level 3 (L3) interconnect
- Capability for the following SoC modules to trigger each other through the embedded cross-trigger system:
  - FPGA fabric
  - A9-0 processor
  - A9-1 processor
  - PTM-0
  - PTM-1
  - STM
  - ETF
  - ETR
  - TPIU
  - csCTI
  - CTI-0
  - CTI-1
  - FPGA-CTI
  - csCTM
  - CTM

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:

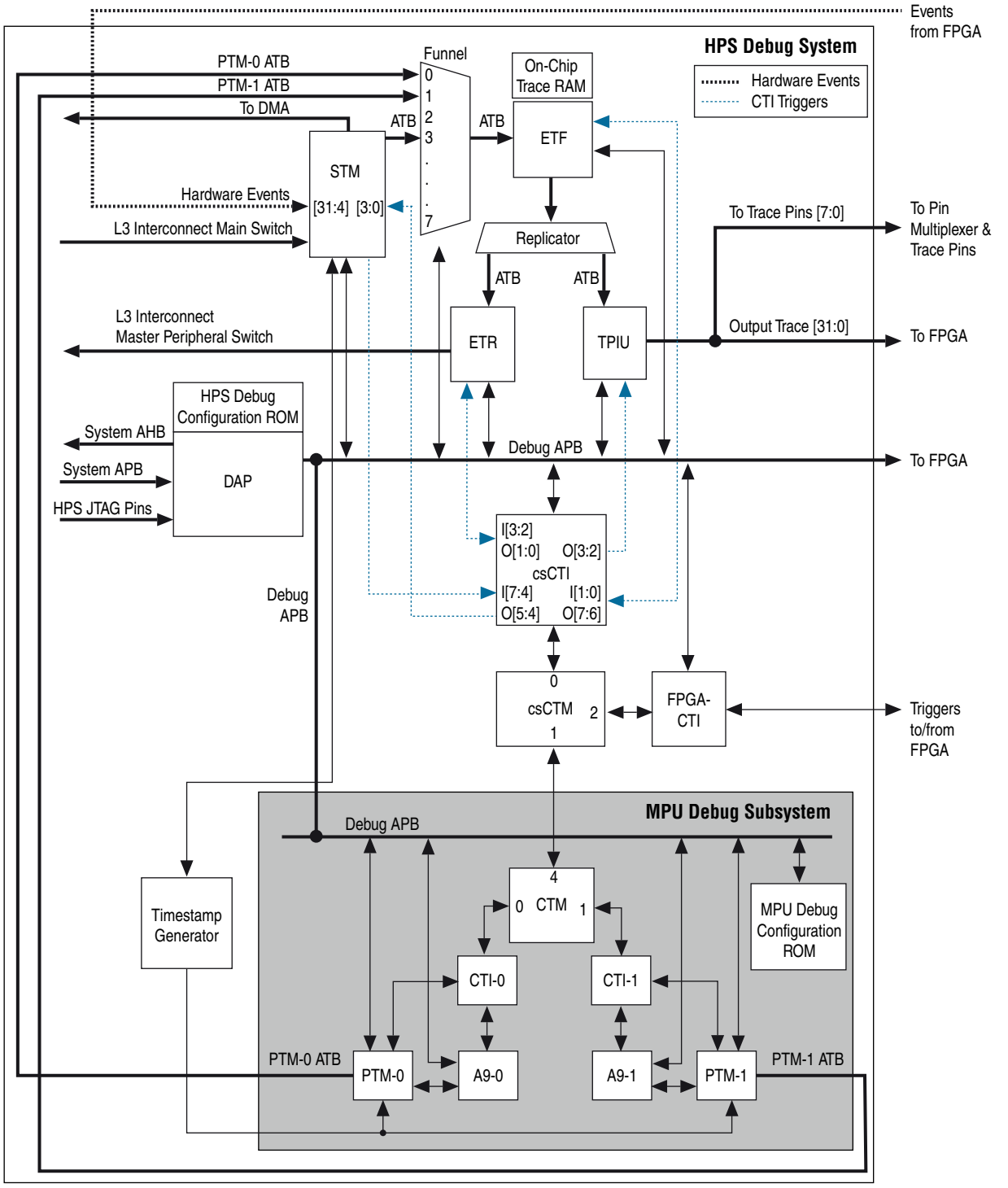
- *CoreSight Technology, System Design Guide, ARM DGI 0012D*
- *CoreSight Architecture Specification, ARM IHI 0029B*
- *ARM Debug Interface v5, Architecture Specification, ARM IHI 0031A*
- *Embedded Cross Trigger Technical Reference Manual, ARM DDI 0291A*
- *CoreSight Components Technical Reference Manual, ARM DDI 0314H*
- *CoreSight Program Flow Trace, Architecture Specification, ARM IHI 0035A*
- *CoreSight PTM-A9 Technical Reference Manual, ARM DDI 0401B*
- *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*

You can download the documents from the ARM website (infocenter.arm.com).

CoreSight Debug and Trace Block Diagram and System Integration

Figure 7–1 shows an overview block diagram of the HPS CoreSight debug and trace system.

Figure 7–1. Debug System Block Diagram



Functional Description of CoreSight Debug and Trace

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
- Multisource trace in a single stream
- Standard programming models for standard tool support

For more information about the CoreSight technology, refer to the *CoreSight Components Technical Reference Manual* and the *CoreSight Technology System Design Guide*, which you can download from the ARM website (infocenter.arm.com).

The following sections provide brief descriptions of ARM CoreSight components provided in the HPS debug system.

Debug Access Port (DAP)

The 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. As shown in Figure 7–1, all CoreSight components are connected to the debug APB.

A host debugger can access any HPS memory-mapped resource in the system via the DAP system master port. Requests made over the DAP system master port are impacted by reads and writes to peripheral registers.

For more information, refer to the *CoreSight Components Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

System Trace Macrocell (STM)

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 via stimulus ports or the hardware event interface. The STM allows the 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 via 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.

For more information, refer to the *CoreSight System Trace Macrocell Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

Trace Funnel

The Trace Funnel multiplexes three trace sources into a single trace stream. Port 0 of the Trace Funnel is connected to the PTM for CPU 0. Port 1 of the Trace Funnel is connected to the PTM for CPU 1. Port 3 of the Trace Funnel is connected to the STM. Port 2 and Port 4 through Port 7 are not used.

For more information, refer to the *CoreSight Components Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

Embedded Trace FIFO (ETF)

The output of the Trace Funnel 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.

AMBA Trace Bus Replicator (Replicator)

The Replicator broadcasts trace data from the ETF to the embedded trace router (ETR) and trace port interface unit (TPIU).

For more information, refer to the *CoreSight Components Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

Embedded Trace Router (ETR)

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 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.

Trace Port Interface Unit (TPIU)

The TPIU is a bridge between on-chip trace sources and an off-chip trace port. The TPIU receives trace data from the 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.

For more information, refer to the *CoreSight Components Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

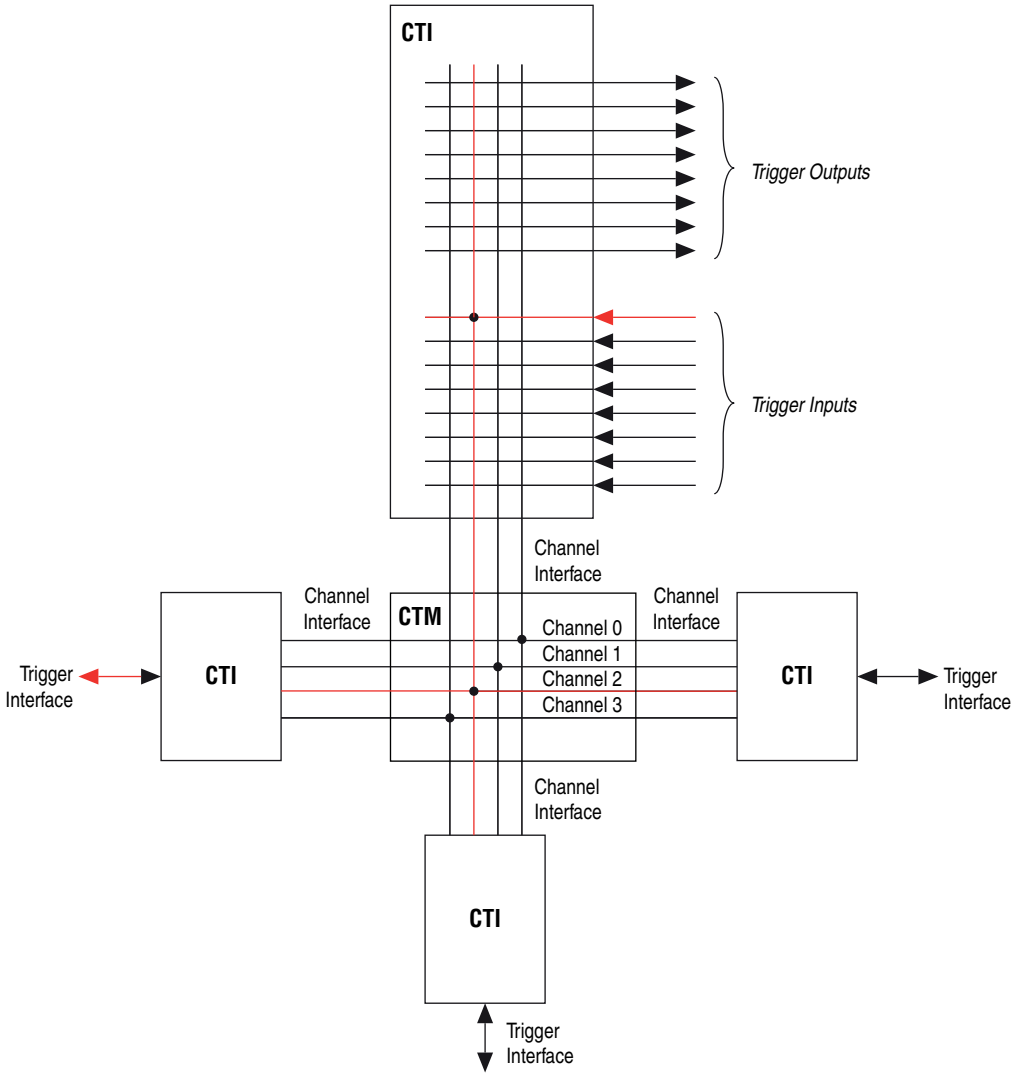
Embedded Cross Trigger (ECT) System

The ECT system provides a mechanism for HPS modules to trigger each other. The ECT consists of the following modules:

- Cross Trigger Interface (CTI)
- Cross Trigger Matrix (CTM)

Figure 7–2 shows how CTIs and CTMs are used in a generic ECT setup. The red line depicts an trigger input to one CTI generating a trigger output in another CTI. Though the signal travels throughout channel 2, it only enters and exits through trigger inputs and outputs you configure.

Figure 7–2. Generic ECT System

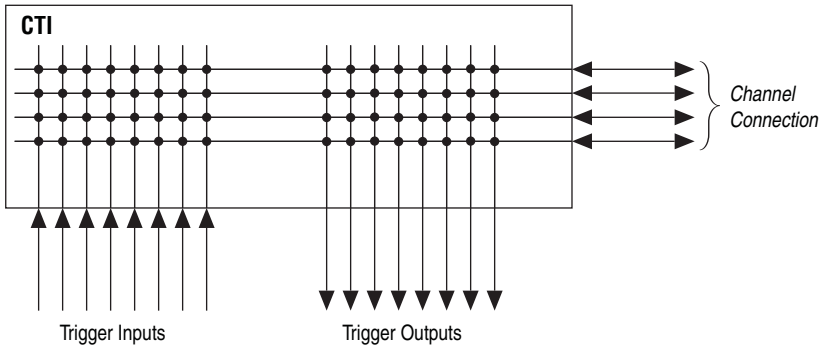


Cross Trigger Interface (CTI)

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 7–2 shows the relationship of trigger inputs, trigger outputs, and CTM channels of a CTI.

Figure 7–3 shows trigger input and trigger output connections in detail.

Figure 7–3. CTI Connections



The HPS debug system contains the following four 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 associated PTM.

Cross Trigger Matrix (CTM)

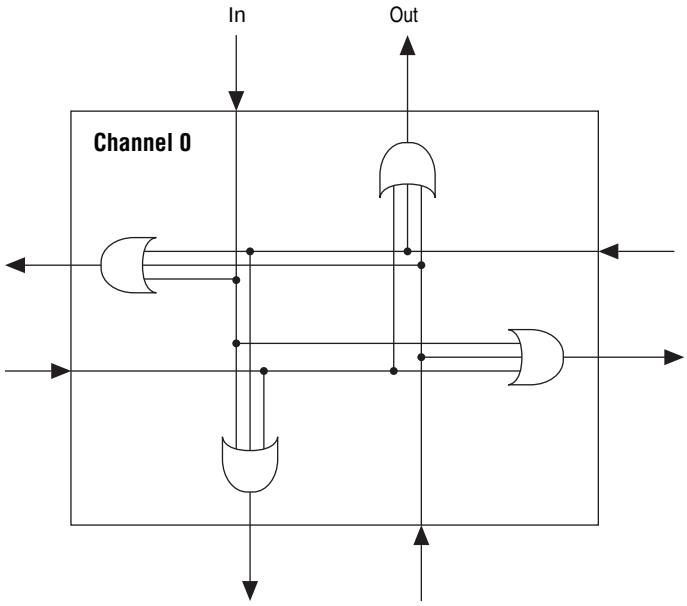
A CTM is a transport mechanism for triggers traveling from one CTI to one or more CTIs or CTMs. The HPS contains two CTMs. One CTM connects csCTI and FPGA-CTI; the other connects CTI-0 and CTI-1. The two CTMs are connected together, allowing triggers to be transmitted between the MPU debug subsystem, 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 7–4 shows the structure of a CTM channel. Paths inside the CTM are purely combinatorial.

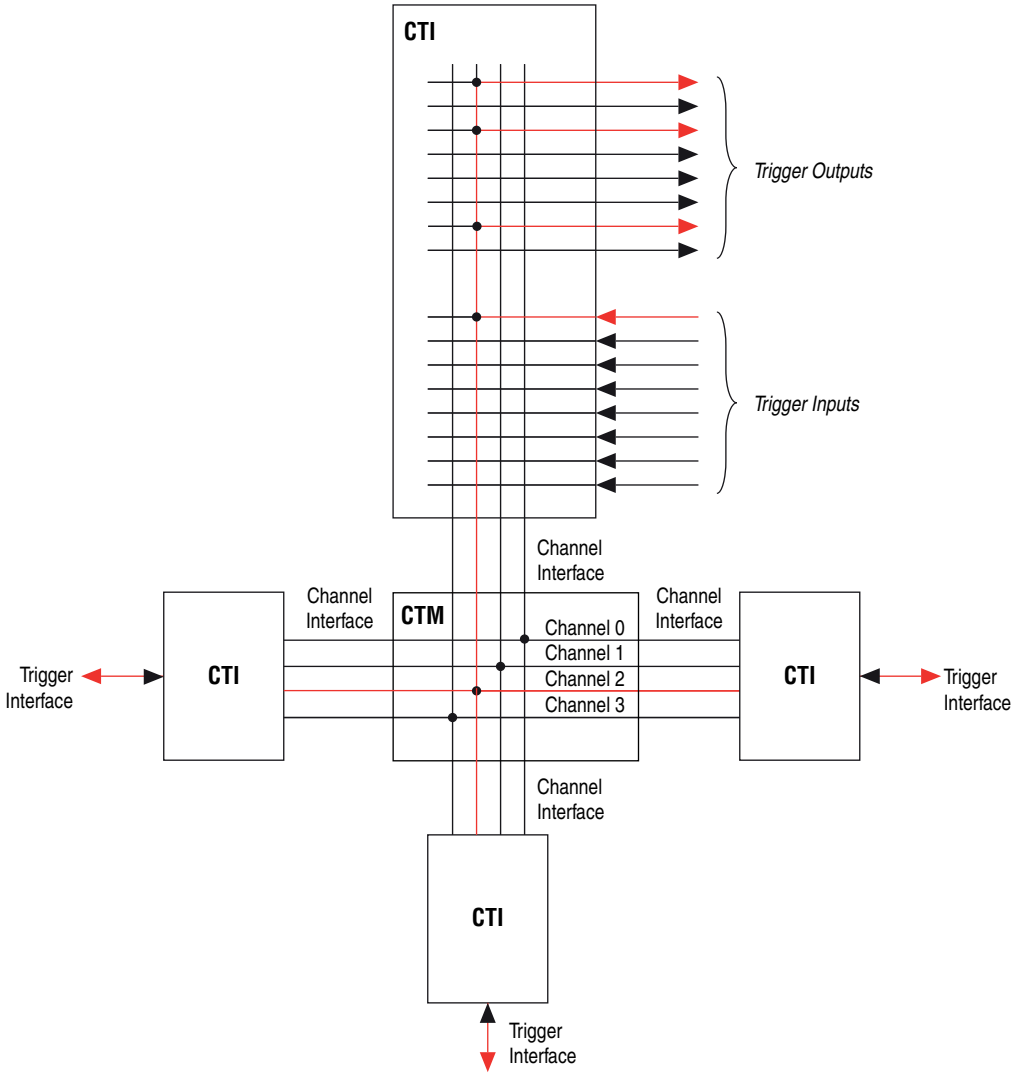
Each CTI trigger input can be connected through a CTM to one or more trigger

Figure 7–4. CTM Channel Structure



outputs under control by a debugger. Figure 7–5 shows a pictorial representation of CTI trigger connections. The red lines depict the impact one trigger input can have on the entire system.

Figure 7–5. CTI Trigger Connections



For more information, refer to the *CoreSight Components Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

Program Trace Macrocell (PTM)

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.

For more information, refer to the *CoreSight PTM-A9 Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

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.

For more information, refer to the *HPS Component Interfaces* chapter in volume 3 of the *Cyclone V Device Handbook*.

CoreSight Debug and Trace Programming Model

This section describes programming model details specific to Altera's implementation of the ARM CoreSight technology.

For programming interface details of each CoreSight component, refer to the *CoreSight Components Technical Reference Manual*, which you can download from the ARM website (infocenter.arm.com).

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. 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 a ROM table.

ROM Table

Table 7–1 contains entries found in the ROM table portion of the DAP.

Table 7–1. DAP ROM Table

ROM Entry	Offset[30:12]	Description
0x0	0x00001	ETF
0x1	0x00002	CTI
0x2	0x00003	TPIU
0x3	0x00004	Trace Funnel
0x4	0x00005	STM
0x5	0x00006	ETR
0x6	0x00007	FPGA-CTI
0x7	0x00100	A9ROM
0x8	0x00080	FPGAROM
0x9	0x00000	End of ROM

A host debugger can access this table at 0x8000_0000 through the DAP. HPS masters can access this ROM at 0xFF00_0000. 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 0xFF00_0000). For example, the CTI output enable register, CTIOUTEN[2] at offset 0xA8, can be accessed by the host debugger at 0x8000_20A8. To derive that value, add the host debugger access address to the ROM table of 0x8000_0000, to the CTI component base address of 0x0000_2000, 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 7–2 lists the address allocation for the STM address segments.

Table 7–2. STM AXI Slave Port Address Allocation

Segment	Start Address	End Address
0	0xFC00_0000	0xFCFF_FFFF
1	0xFD00_0000	0xFDFF_FFFF
2	0xFE00_0000	0xFEFF_FFFF

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 7–3 lists the fields of the STM address.

Table 7–3. STM AXI Address Fields

AXI Signal Fields	Description
AWADDRS[7:0]	These bits index the 256 bytes of the stimulus port. For more information, refer to the *System Trace Macrocell, Programmers' Model Architecture Specification*, which you can download from the ARM website (infocenter.arm.com).
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 0xFC00_0000 through 0xFEFF_FFFF.

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. Table 7–4 shows how the STM master ID is calculated.

Table 7–4. 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 seven-bit master ID.
Master ID[6]	AWPROT[1]	0 indicates secure; 1 indicates nonsecure.

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.

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 7–5 lists the trigger input pin connections implemented for csCTI.

Table 7–5. Trigger Input Signals

Number	Signal	Source
7	ASYNCOUT	STM
6	TRIGOUTHETE	STM
5	TRIGOUTSW	STM
4	TRIGOUTSPTE	STM
3	ACQCOMP	ETR
2	FULL	ETR
1	ACQCOMP	ETF
0	FULL	ETF

Table 7–6 lists the trigger output pin connections implemented for csCTI.

Table 7–6. Trigger Output Signals

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 7–7 lists the trigger output acknowledge pin connections implemented for csCTI.

Table 7–7. Trigger Output Acknowledge Signals

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.

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 website (infocenter.arm.com).

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 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.

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. Refer to Figure 7–1.

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 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.

Debug Clocks

The CoreSight system uses several different clocks. Table 7–8 provides a list of these clocks. Port Name is the name of the clock signal inputs described for individual CoreSight debug components in the ARM documentation. Signal Name is the name of the clock signal used with other HPS components.

Table 7–8. CoreSight Clocks

Port Name	Clock Source	Signal Name	Description
ATCLK	Clock manager	dbg_at_clk	Trace bus clock.
CTICLK (for csCTI)	Clock manager	dbg_at_clk	Cross trigger interface clock for csCTI. It can be synchronous or asynchronous to CTMCLK.
CTICLK (for FPGA-CTI)	FPGA fabric	fpga_cti_clk	Cross trigger interface clock for FPGA-CTI.
CTICLK (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 CTICLK.
CTMCLK (for CTM)	Clock manager	mpu_clk	Cross trigger matrix clock for CTM. It can be synchronous or asynchronous to CTICLK.
DAPCLK	Clock manager	dbg_clk	DAP internal clock. It must be equivalent to PCLKDBG.
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.
PCLKSYS	Clock manager	l4_mp_clk	Used by the APB slave port inside the DAP. It is asynchronous to DAPCLK.
SWCLKTCK	JTAG interface	dap_tck	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.
	FPGA fabric	tpiu_traceclkin	
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.

For more information about the CoreSight port names, refer to table 6-2 in the *CoreSight Technology System Design Guide*, which you can download from the ARM website (infocenter.arm.com).

Debug Resets

The CoreSight system uses several resets. Table 7–9 provides a list of these resets. Port Name is the name of the clock signal inputs described for individual CoreSight debug components in the ARM documentation. Signal Name is the name of the clock signal used with other HPS components.

Table 7–9. CoreSight Resets

Port Name	Clock Source	Signal Name	Description
ATRESETn	Reset manager	dbg_rst_n	Trace bus reset. It resets all registers in ATCLK domain.
nCTIRESET	Reset manager	dbg_rst_n	CTI reset signal. It resets all registers in 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.
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.

For more information about the CoreSight port names, refer to table 6-3 in the *CoreSight Technology System Design Guide*, which you can download from the ARM website (infocenter.arm.com).

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.

For more information about reset handshaking, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

The level 4 (L4) watchdog timers can be paused during debugging to prevent reset while the processor is stopped at a breakpoint.

For more information, refer to the *Watchdog Timer* chapter in volume 3 of the *Cyclone V Device Handbook*.

CoreSight Debug and Trace Address Map and Register Definitions

The address map resides in the **hps.html** file that accompanies this handbook volume. The register definitions reside in separate ARM documentation. Click the link to open the file.

To view the debug-related module descriptions and base addresses, scroll to and click the following links:

- **stm**
- **dap**
- **dmanonsecure**
- **dmasecure**
- **mpuscu**
- **mpul2**

To then view the register and field descriptions, click the link in the module description to access the appropriate ARM documentation. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 7–10 shows the revision history for this document.

Table 7–10. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
June 2012	1.1	Added functional description, programming model, and address map and register definitions sections.
January 2012	1.0	Initial release.

Section V. Memory and Memory Controllers

This section includes the following chapters:

- Chapter 8, SDRAM Controller Subsystem
- Chapter 9, On-Chip Memory
- Chapter 10, NAND Flash Controller
- Chapter 11, SD/MMC Controller
- Chapter 12, Quad SPI Flash Controller

For information about the revision history for chapters in this section, refer to "Document Revision History" in each individual chapter.

8. SDRAM Controller Subsystem

cv_54008-1.1

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 the following features:

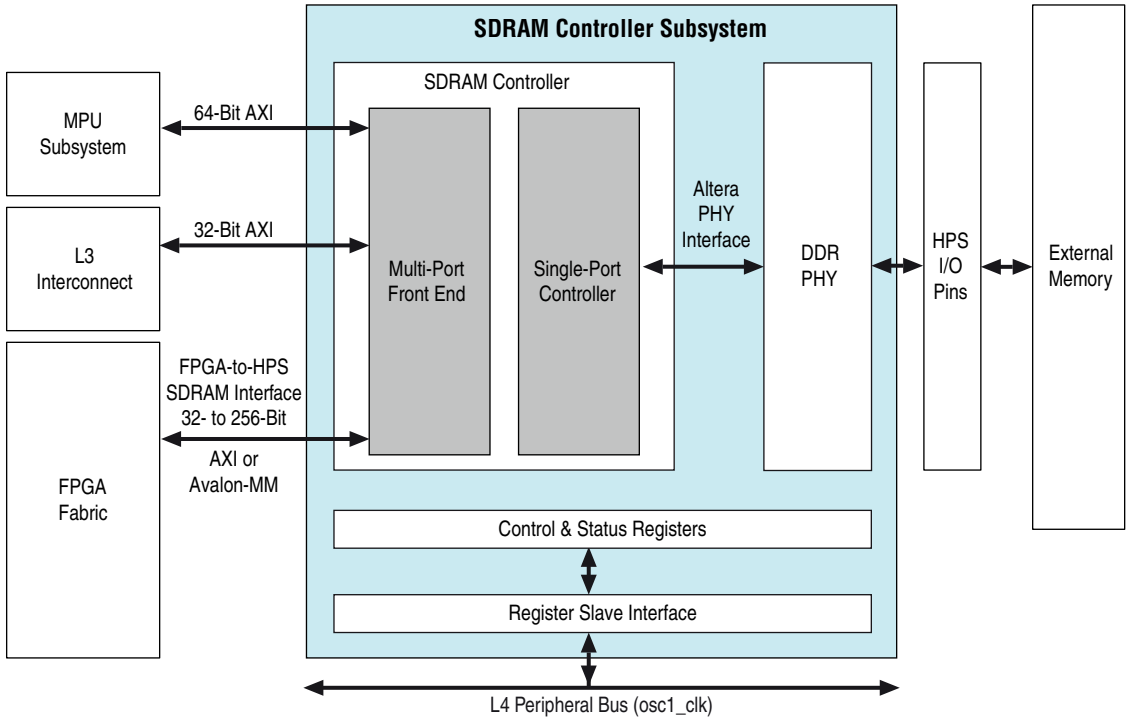
- Support for double data rate 2 (DDR2), DDR3, and low-power DDR2 (LPDDR2) SDRAM
- User-configurable timing parameters
- Up to 4 Gb density parts
- Two chip selects
- Integrated error correction code (ECC), 24- and 40-bit widths
- User-configurable memory width of 8, 16, 16+ECC, 32, 32+ECC
- Command 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 relative priority scheduling
- Flexible FPGA fabric interface configuration with up to 6 ports and data widths up to 256 bits wide using Avalon-MM and AXI interfaces.
- Power management supporting self refresh, partial array self-refresh (PASR), power down, and LPDDR2 deep power down

SDRAM Controller Subsystem Block Diagram and System Integration

The SDRAM controller subsystem connects to the MPU subsystem, the main switch of 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 8–1 shows a high-level block diagram of the SDRAM controller subsystem.

Figure 8–1. 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. For more information, refer to "Memory Controller Architecture" on page 8–4.

DDR PHY

The DDR PHY provides a physical layer interface between the memory controller and memory devices, which performs read and write memory operations. The DDR PHY has dataflow components, control components, and calibration logic that handle the calibration for the SDRAM interface timing.

SDRAM Controller Subsystem Interfaces

The following sections describe the SDRAM controller subsystem interfaces.

MPU Subsystem Interface

The SDRAM controller is connected 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 is connected to the L3 interconnect with a dedicated 32-bit AXI interface, operating on the `l3_main_clk` clock domain.

CSR Interface

The CSR interface is connected 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 to 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 ports 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 8–1 lists the FPGA-to-HPS SDRAM controller interface ports connected to the FPGA.

Table 8–1. FPGA-to-HPS SDRAM Controller Port Types

Port Type	Number
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.

- 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.

Table 8–2 shows the number of ports needed to configure different bus protocols, based on type and data width.

Table 8–2. FPGA-to-HPS SDRAM Port Utilization

Bus Protocol	Command	Read Data	Write Data
32- or 64-bit AXI	2 (1)	1	1
128-bit AXI	2 (1)	2 (2)	2 (2)
256-bit AXI	2 (1)	4 (2)	4 (2)
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

Notes to Table 8–2:

(1) Because the AXI protocol allows simultaneous read and write commands to be issued, two SDRAM control ports are required to form an AXI interface.

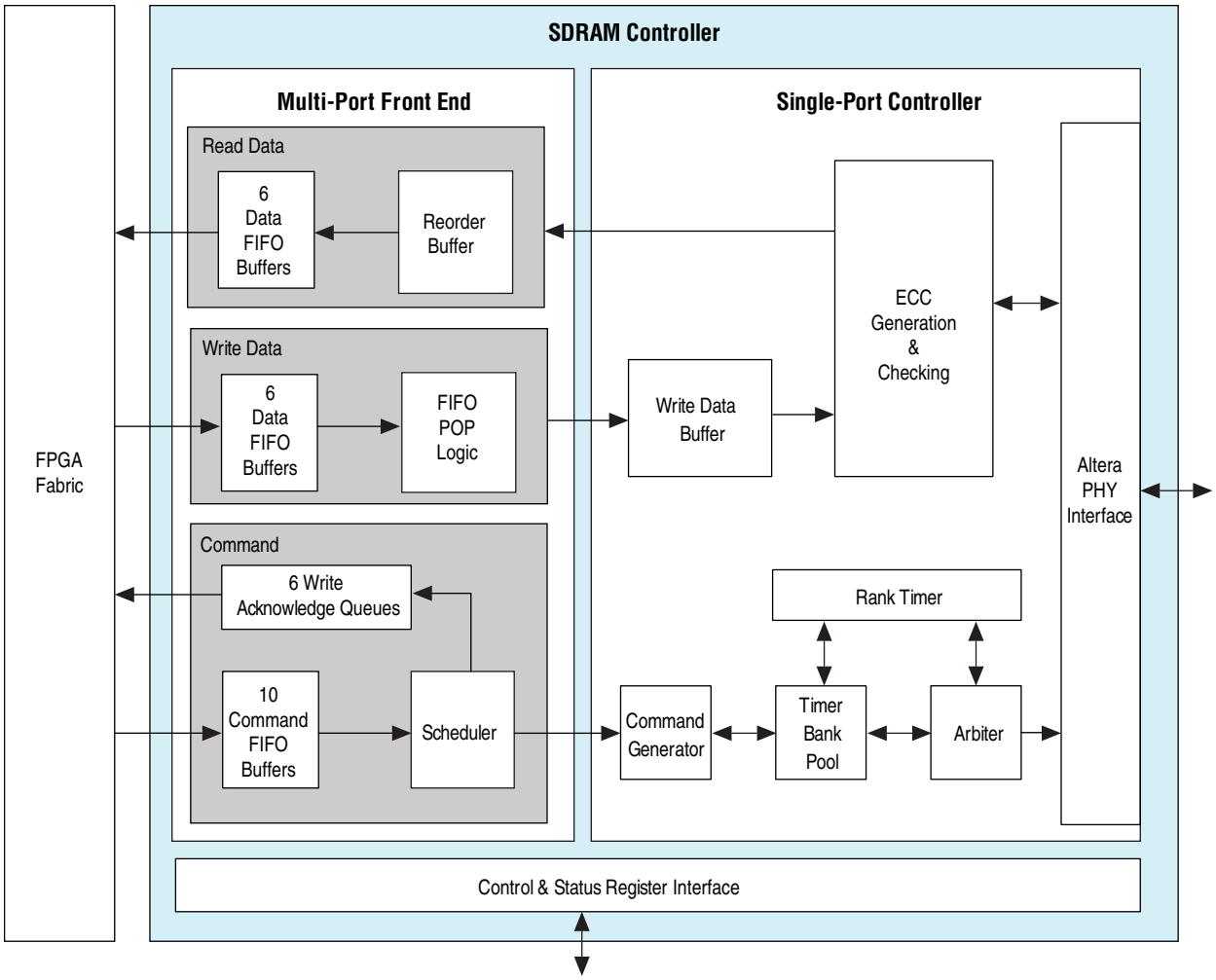
(2) Because the native size of the data ports is 64 bits, extra read and write ports are required to form an AXI interface.

Memory Controller Architecture

The SDRAM controller consists of an MPFE, a single-port controller, and an interface to the CSRs.

Figure 8–2 shows a block diagram of the SDRAM controller portion of the SDRAM controller subsystem.

Figure 8–2. SDRAM Controller Block Diagram



MPFE

The 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 the following 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 multiport logic).

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.

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.

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.

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

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.

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.

AFI Interface

The AFI interface provides communication between the controller and the PHY.

CSR Interface

The CSR interface is accessible from the L4 bus. The interface allows code executing in the HPS MPU and FPGA fabric to configure and monitor the SDRAM controller.

Functional Description of the SDRAM Controller Subsystem

This section provides a functional description of the SDRAM controller subsystem.

MPFE Operational Behavior

This section describes the operational behavior of the MPFE.

Operation Ordering

Requests to the same SDRAM page arriving at a given port are executed in the order in which they are received. Requests arriving at different ports have no guaranteed order of service, except when a first transaction has completed before the second arrives.

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. Transactions received on different ports have no guaranteed order unless the second transaction is presented after the first has completed.

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 ensure that the overall latency of traffic is as low as possible, the single port logic can return read data out of order to the multi-port logic which will reorder it when transactions return out of order. A large percentage of traffic reordering will be between ports and transactions only are ordered within a port. For traffic which is reordered between ports but not within a port, no reordering needs to be done. Eliminating unnecessary reordering reduces average latency.

Multiport Scheduling

Multiport scheduling is governed by two factors, the absolute priority of a request and the weighting of a port.

The evaluation of absolute priority ensures that ports carrying higher-priority traffic are served ahead of ports carrying lower-priority traffic. The scheduler recognizes eight priority levels (0-7), with higher values representing higher priorities. For example, any transaction with priority seven is scheduled before transactions of priority six or lower.

When ports carry traffic of the same absolute priority, relative priority is determined based on port weighting. Port weighting is a five-bit value (0-31), and is determined by a deficit-weighted round robin (DWRR) algorithm, which corrects for past over-servicing or under-servicing of a port. Each port has an associated weight which is updated every cycle, with a user-configured weight added to it and the amount of traffic served subtracted from it. The port with the highest weighting is considered the most eligible.

To ensure that high-priority traffic is served quickly and that long and short bursts are effectively interleaved between ports, incoming transactions longer than a single SDRAM burst are scheduled as a series of SDRAM bursts, with each burst arbitrated separately.

To ensure that lower priority ports do not build up large running weights while higher priority ports monopolize bandwidth, the controller's DWRR weights are updated only when a port matches the scheduled priority. Therefore, if three ports are being accessed, two being priority seven and one being priority four, the weights for both ports at priority seven are updated but the port with priority four remains unchanged.

Multiport scheduling is performed between all of the ports connected to the FPGA fabric and internally in the HPS to determine which transaction is serviced next. Arbitration is performed on a SDRAM burst basis to ensure that a long transaction does not lock other transactions or cause latency to significantly increase for high-priority ports.

Arbitration supports both absolute and relative priority. Absolute priority is intended for applications where one master should always get priority above or below others. Relative priority is supported through a programmable weight field which controls scheduling between ports at the same priority.

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.

The multiport scheduling configuration can be updated while traffic is flowing. Both priority and weight for a port can be updated without interrupting traffic on a port. Updates are used in scheduling decisions within 10 memory clock cycles of being updated, so priority can be updated frequently if needed.

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.

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. For example, 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 Operational Behavior

This section describes the operational behavior of the single-port controller.

SDRAM Interface

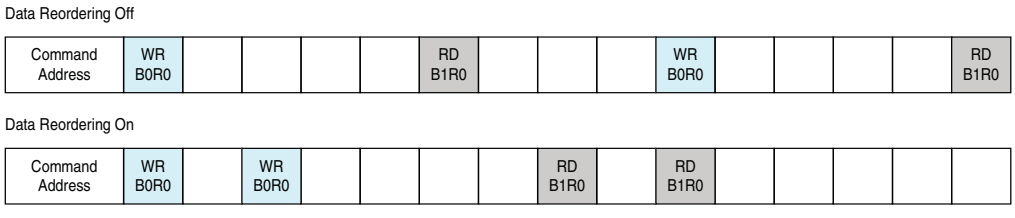
The SDRAM interface is up to 40 bits wide and can accommodate 8-bit, 16-bit, 16-bit plus ECC, 32-bit, or 32-bit plus ECC configurations. The SDRAM interface supports LPDDR2, DDR2, and DDR3 memory protocols.

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.

Figure 8–3 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 8–3. Data Reordering Effect



The SDRAM controller schedules among all pending row and column commands every clock cycle.

Bank Policy

The bank policy of the SDRAM controller allows users to request that a transaction's 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 will cause 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, and data widths of 8, 16, and 32 bits for non-ECC operation, and widths of 24 and 40 operations with ECC enabled. Table 8–3 shows the type of SDRAM for each burst length.

Table 8–3. SDRAM Burst Lengths

Burst Length	SDRAM
4	LPDDR2, DDR2
8	DDR2, DDR3, LPDDR2
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.

Controller ECC provides the following features:

- 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. 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—When a read operation detects a correctable error, the memory location is scheduled for a read-modify-write operation to correct the single-bit error. ECC write backs are enabled and disabled through the `cfg_enable_ecc_code_overwrites` field in the `ctrlcfg` register.
- Notification of ECC errors—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.

Byte Writes

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), then no read is necessary and only that word is updated.

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.

User Notification of ECC Errors

The following methods notify you of an ECC error:

For the MPU subsystem, an interrupt signal provides notification and the ECC error information is stored in the status registers.

For more information, refer to the *Cortex-A9 Microprocessor Unit SubSystem* chapter in volume 3 of the *Cyclone V Device Handbook*.

Interleaving Options

The controller supports the following address-interleaving options:

- Noninterleaved
- Bank interleave without chip select interleave
- Bank interleave with chip select interleave

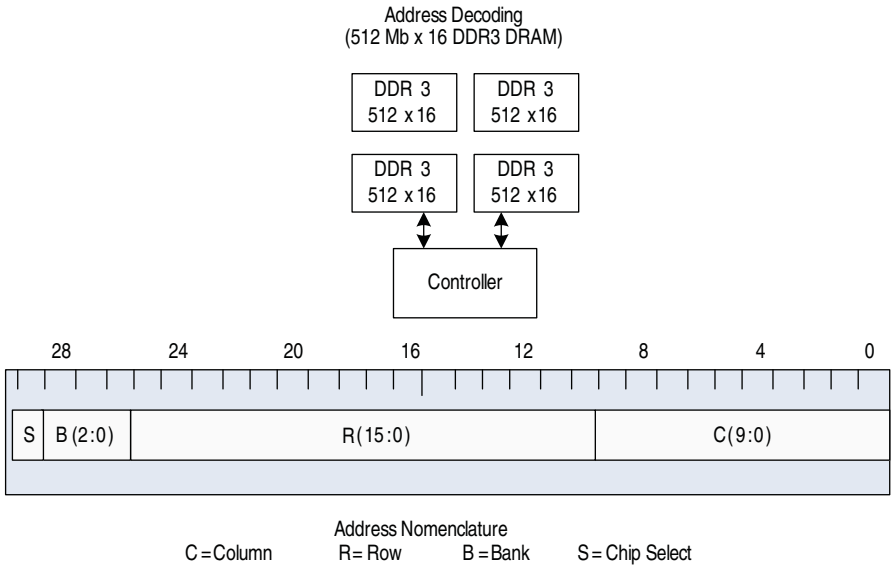
All of the interleaving examples use 512 megabits (Mb) x 16 DDR3 chips and are documented as byte addresses. For RAMs with smaller address fields, the order of the fields stays the same but the widths may change.

Noninterleaved

RAM mapping is noninterleaved.

Figure 8–4 shows noninterleaved address decoding.

Figure 8–4. Noninterleaved Address Decoding

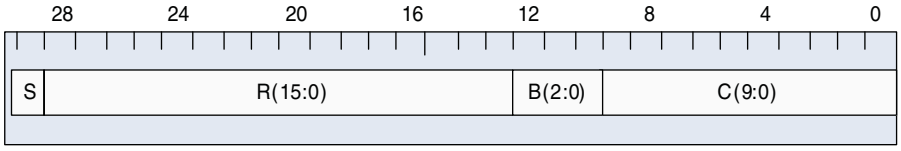


Bank Interleave Without Chip Select Interleave

Bank interleave without chip select interleave swaps row and bank from the noninterleaved address mapping. This interleaving allows smaller data structures to spread across all banks in a chip.

Figure 8–5 shows bank interleave without chip select interleave address decoding.

Figure 8–5. Bank Interleave Without Chip Select Interleave Address Decoding

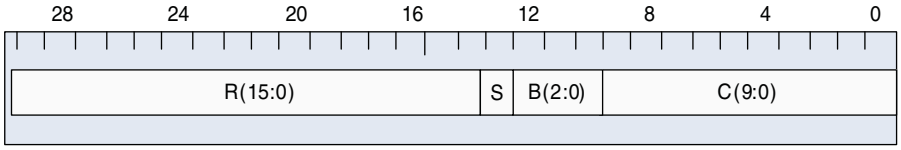


Bank Interleave with Chip Select Interleave

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 8–6 shows bank interleave with chip select interleave address decoding.

Figure 8–6. 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.

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 AXI masters, TrustZone® is supported. Ports that use Avalon-MM can be configured for port level protection.

For information about TrustZone®, refer to the ARM website (www.arm.com).

Memory protection is based on physical addresses in memory. You can set rules to allow or disallow accesses to a range of memory, or to enable only secure accesses to a range of memory (or a combination of the two).

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 memory protection table, which is an internal table addressed through the CSR interface, contains rules to permit or deny memory access. You can configure up to a maximum of twenty rules to control memory access. Table 8–4 lists the fields that you can specify for each rule.

Table 8–4. Fields for Rules in Memory Protection Table (Part 1 of 2)

Field	Width	Description
Valid	1	Set to 1 to activate the rule. Set to 0 to deactivate the rule.
Port Mask (1)	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 switch read, bit 7 is the CPU read, bit 8 is L3 switch write, and bit 9 is the CPU write.

Table 8–4. Fields for Rules in Memory Protection Table (Part 2 of 2)

Field	Width	Description
TID_low (1)	12	Low transfer ID of the rules to which this rule applies. Incoming transactions match if they are greater than or equal to this value. Ports with smaller TIDs have the TID shifted to the lower bits and zero padded at the top.
TID_high (1)	12	High transfer ID 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.
Protection	2	A value of 00 indicates that the protection bit is not set; a value of 01 sets the protection bit. Systems that do not set AXI protection to a known value should program this for either protection value.
Fail/allow	1	Set this value to 1 to force the operation to fail or succeed.

Note to Table 8–4:

(1) Although TID 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 TIDs, a port-based rule might be more efficient than a TID-based rule, in terms of the number of rules needed.

A port has a default access status of either allow or fail, and rules with the opposite allow/fail value can override the default. The system evaluates each transaction against every rule in the memory protection table. A transaction received on a port which by default allows access, would fail only if a rule with the fail bit matches the transaction. Conversely, a port which by default prevents access, would allow access only if a rule allows that transaction to pass.

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.

Example of Configuration for TrustZone

For a TrustZone configuration, memory is divided into a range of memory accessible by secure masters and a range of memory accessible by nonsecure 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 nonsecure area

In this example, each port is configured by default to disallow all accesses. Table 8–5 shows the two rules programmed into the memory protection table.

Table 8–5. Rules in Memory Protection Table for Example Configuration

Rule #	Port Mask	TID Low	TID High	Address Low	Address High	Prot	Fail/Allow
1	0'b1111111111	0	4095	0	576	b01	allow
2	0'b1111111111	0	4095	512	2047	b00	allow

The port mask value, TID Low, and TID High, apply to all ports and all transfers within those ports. Each access request is evaluated against the memory protection table, and will fail unless a rule matches allowing a transaction to complete successfully.

Table 8–6 shows the result for a sample set of transactions.

Table 8–6. Result for a Sample Set of Transactions

Operation	Source	Address	Prot	Result	Comments
Read	CPU	4096	1	Allow	Matches rule 1.
Write	CPU	536, 870, 912 (512 MB)	1	Allow	Matches rule 1.
Write	L3 attached masters	605, 028, 350 (577 MB)	1	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	0	Fail	Does not match rule 1 (prot value wrong), does not match rule 2 (not in address range).
Write	CPU	536, 870, 912 (512 MB)	0	Allow	Matches rule 2.
Write	L3 attached masters	605, 028, 350 (577 MB)	0	Allow	Matches rule 2.

If you did not want any overlap between the memory blocks, you could specify the address ranges in the two rules of Table 8–5 to be mutually exclusive. Depending on your desired TrustZone configuration, you can add rules to the memory protection table to create multiple blocks of protected or unprotected space.

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

Power-saving mode initiates either due to a user command or from inactivity.

Power-down mode is initiated by writing to the appropriate control register. It forces the SDRAM burst-scheduling bank-management logic to close all banks and issue the power down command. You can program the controller to enable power-down when the SDRAM burst-scheduling queue is empty for a specified number of clock cycles. The SDRAM automatically reactivates when an active SDRAM command is received.

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.

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 8–7 shows the SDRAM controller subsystem clock domains.

Table 8–7. SDRAM Controller Subsystem Clock Domains

Clock Name	Description
`ddr_dq_clk`	Clock for PHY
`ddr_dqs_clk`	Clock for MPFE, single-port controller, CSR access, and PHY
`ddr_2x_dqs_clk`	Clock for PHY
`l4_sp_clk`	Clock for CSR interface
`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.

For more information, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Resets

The SDRAM controller subsystem supports a full reset (cold reset) and a warm reset, which may or may not preserve the contents of memory. In order to preserve the 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 memory contents are preserved, the PHY and the memory timing logic is not reset, but the rest of the controller is reset.

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Initialization

The SDRAM controller subsystem has 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 Qsys.

The CSRs are configured using a dedicated slave interface, which provides accesses to 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.

Protocol Details

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 Table 8–8.

Table 8–8. Avalon-MM Bidirectional Port Signals (Part 1 of 2)

Name	Bits	Direction	Function
clk	1	In	Clock for the Avalon-MM interface
read	1	In	Indicates read transaction
write	1	In	Indicates write transaction
address	32	In	Address of the transaction
readdata	32, 64, 128, or 256	Out	Read data return
readdatavalid	1	Out	Valid cycle flag for read data return
writedata	32, 64, 128, or 256	In	Write data for a transaction
byteenable	4 (32-bit data), 8(64-bit data), 16(128-bit data), 32(256-bit data)	In	Byte enables for each write byte

Table 8–8. Avalon-MM Bidirectional Port Signals (Part 2 of 2)

Name	Bits	Direction	Function
waitrequest	1	Out	Indicates need for additional cycles to complete a transaction
burstcount	11	In	Transaction burst length

The read and write interfaces are configured to the same size. The byte-enable size scales with the data bus size.

For information about the Avalon-MM protocol, refer to the *Avalon Interface Specifications*.

Avalon-MM Write Port

The Avalon-MM write ports are standard Avalon-MM ports used only to dispatch write operations. Each configured Avalon-MM write port consists of the signals listed in Table 8–9.

Table 8–9. Avalon-MM Write 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 (32-bit data), 8(64-bit data), 16(128-bit data), 32(256-bit data)	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

For information about the Avalon-MM protocol, refer to the *Avalon Interface Specifications*.

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 Table 8–10.

Table 8–10. Avalon-MM Read Port Signals (Part 1 of 2)

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

Table 8–10. Avalon-MM Read Port Signals (Part 2 of 2)

Name	Bits	Direction	Function
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

For information about the Avalon-MM protocol, refer to the *Avalon Interface Specifications*.

AXI Port

The AXI port uses an AXI-3 interface.

For information about the AXI-3 interface, refer to the *AMBA Open Specifications* on the ARM website (www.arm.com).

For information about the AXI interface ports in the high-performance II controller (HPC II), refer to the *Functional Description—HPC II Controller* chapter, in the *External Memory Interface Handbook*.

Each configured AXI port consists of the signals listed in Table 8–11. Each AXI interface signal is independent of the other interfaces for all signals, including clock and reset.

Table 8–11. AXI Port Signals (Part 1 of 2)

Name	Bits	Direction	Function
ARESETn	1	In	Reset
ACLK	1	In	Clock
Write Address Channel Signals			
AWID	4	In	Write identification tag
AWADDR	32	In	Write address
AWLEN	4	In	Write burst length
AWSIZE	3	In	Width of the transfer size
AWBURST	2	In	Burst type
AWREADY	1	Out	Indicates ready for a write command
AWVALID	1	In	Indicates valid write command.
Write Data Channel Signals			
WID	4	In	Write data transfer ID
WDATA	32, 64, 128 or 256	In	Write data
WSTRB	4, 8, 16, 32	In	Byte-based write data strobe. Each bit width corresponds to 8 bit wide transfer for 32-bit wide to 256-bit wide transfer.

Table 8–11. AXI Port Signals (Part 2 of 2)

Name	Bits	Direction	Function
WLAST	1	In	Last transfer in a burst
WVALID	1	In	Indicates write data+strobes are valid
WREADY	1	Out	Indicates ready for write data and strobes
Write Response Channel Signals			
BID	4	Out	Write response transfer ID
BRESP	2	Out	Write response status
BVALID	1	Out	Write response valid signal
BREADY	1	In	Write response ready signal
Read Address Channel Signals			
ARID	4	In	Read identification tag
ARADDR	32	In	Read address
ARLEN	4	In	Read burst length
ARSIZE	3	In	Width of the transfer size
ARBURST	2	In	Burst type
ARREADY	1	Out	Indicates ready for a read command
ARVALID	1	In	Indicates valid read command
Read Data Channel Signals			
RID	4	Out	Read data transfer ID
RDATA	32, 64, 128 or 256	Out	Read data
RRESP	2	Out	Read response status
RLAST	1	Out	Last transfer in a burs
RVALID	1	Out	Indicates read data is valid
RREADY	1	In	Read data channel ready signal

SDRAM Controller Subsystem Programming Model

Initialization

SDRAM controller configuration occurs through software programming of the configuration registers using the CSR interface. Initialization of the SDRAM controller has two separate regions with different controls.

Timing Parameters

The SDRAM controller supports a complete set of timing parameters, configurable at run time.

SDRAM Controller Address Map and Register Definitions

The address map resides in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for the following module instance:

- **sdr**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 8–12 shows the revision history for this document.

Table 8–12. Document Revision History

Date	Version	Changes
November 2012	1.1	Added address map and register definitions section.
January 2012	1.0	Initial release.

9. On-Chip Memory

cv_54009-1.1

The hard processor system (HPS) contains the following on-chip memory:

- On-Chip RAM
- Boot ROM

The on-chip RAM provides 64 KB of general-purpose RAM. The boot ROM contains the code required to boot the HPS from cold or warm reset. Both memories connect to the level 3 (L3) interconnect.

On-Chip RAM

This section describes the HPS on-chip RAM.

Features of the On-Chip RAM

The on-chip RAM offers the following features:

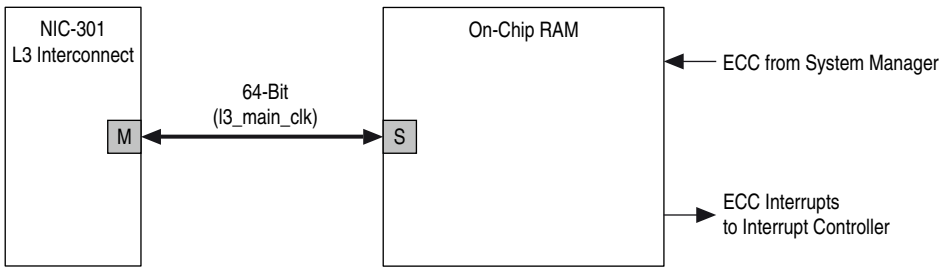
- 64-bit interface
- 64 KB size
- Single-ported RAM
- Read acceptance of two, write acceptance of two, and a total acceptance of two.
- Error correction code (ECC) support
- Sustained ideal throughput (operating frequency times the data width) during read after read, write after write, write after read, and read after write.

For information about the operating frequency, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

On-Chip RAM Block Diagram and System Integration

Figure 9–1 shows a block diagram of the on-chip RAM.

Figure 9–1. On-Chip RAM Block Diagram



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. The memory has a read acceptance of two, a write acceptance of two, and a total acceptance of two with round-robin arbitration.

The entire RAM is either secure or nonsecure. Security is enforced by the NIC-301 L3 interconnect.

For more information about security, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

Functional Description of the On-Chip RAM

The on-chip RAM serves as a general-purpose memory accessible from the FPGA.

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.

Clocks

The on-chip RAM is driven by the `l3_main_clk` interconnect clock.

For information about the operating frequency and maximum throughput, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Resets

The contents of the RAM remain unchanged on a cold or warm reset. Reset only clears the state associated with the slave interface.

The on-chip RAM reset is driven by the `onchip_ram_rst_n` interconnect reset signal.

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Boot ROM

This section describes hardware aspects of the HPS boot ROM.

For information about the boot ROM software, refer to the *Booting and Configuration* appendix in volume 3 of the *Cyclone V Device Handbook*.

Features of the Boot ROM

The boot ROM offers the following features:

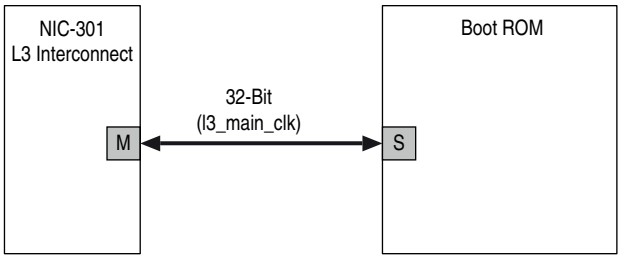
- 32-bit interface
- 64 KB size
- Single-ported ROM
- Read acceptance of two
- Sustained ideal throughput (operating frequency times the data width) during read after read.

For information about the operating frequency, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Boot ROM Block Diagram and System Integration

Figure 9–2 shows a block diagram of the boot ROM.

Figure 9–2. Boot ROM Block Diagram



Transfers between memory and the NIC-301 L3 interconnect happen through a 32-bit 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.

For more information about security, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

Functional Description of the Boot ROM

The boot ROM is used only for booting the system. On a cold or warm reset of the microprocessor unit (MPU) subsystem, MPU0 executes the pre-bootloader code stored in the boot ROM.

For information about the boot ROM software, refer to the *Booting and Configuration* appendix in volume 3 of the *Cyclone V Device Handbook*.

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.

Clocks

The boot ROM is driven by the `l3_main_clk` interconnect clock.

For information about the operating frequency and maximum throughput, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Resets

The contents of the ROM remain unchanged on a cold or warm reset. Reset only clears the state associated with the slave interface.

The boot ROM reset is driven by the `boot_rom_rst_n` interconnect clock.

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

On-Chip Memory Address Map and Register Definitions

There are no registers for on-chip memory.

The address map resides in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module descriptions and base addresses, scroll to and click the links for the following module instances:

- **rom**
- **ocram**

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 9–1 shows the revision history for this document.

Table 9–1. Document Revision History

Date	Version	Changes
November 2012	1.1	Added address map section.
January 2012	1.0	Initial release.

10. NAND Flash Controller

cv_54010-1.2

The hard processor system (HPS) provides a NAND flash controller to interface with external NAND flash memory in Altera® system-on-a-chip (SoC) FPGA systems. You can use external flash memory to store a processor boot image, software, or as extra storage capacity for large applications or user data. The HPS NAND flash controller is based on the Cadence® Design IP® NAND Flash Memory Controller.

NAND Flash Controller Features

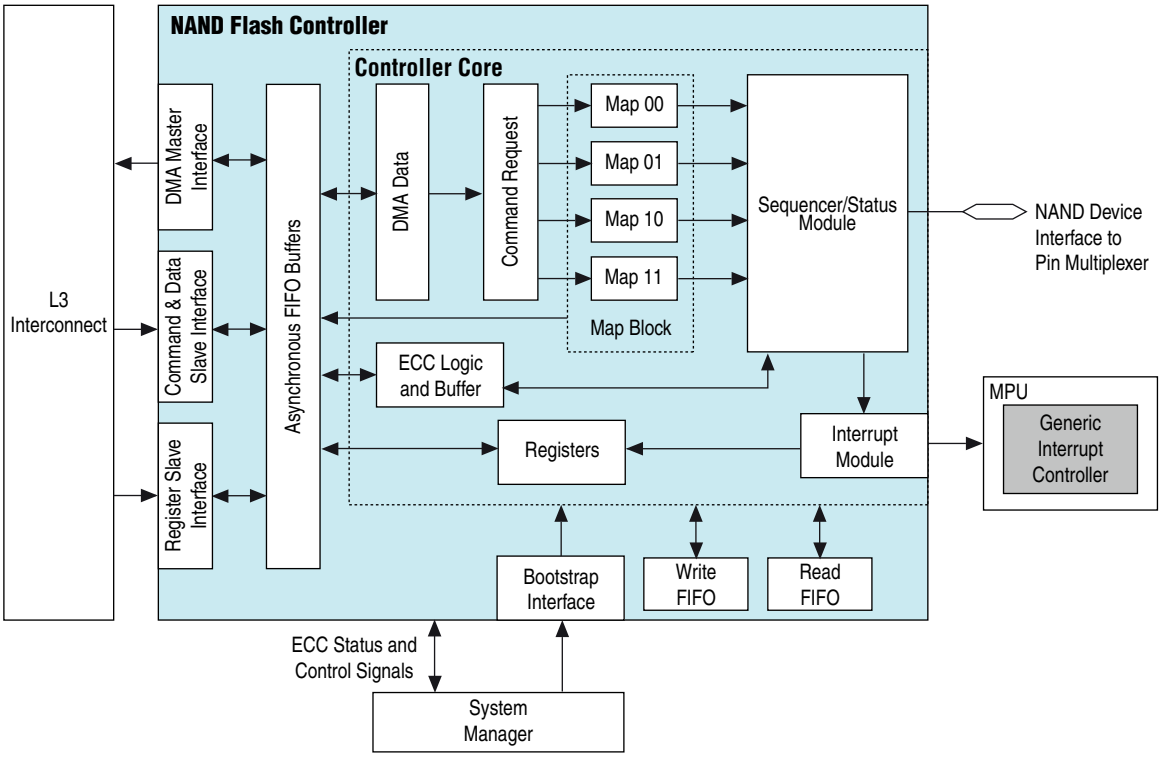
The NAND flash controller provides the following functionality and features:

- Supports one x8 NAND flash device
- Supports Open NAND Flash Interface (ONFI) 1.0
- Supports NAND flash memories from Hynix, Samsung, Toshiba, Micron, and ST Micro
- Supports programmable 512 byte (4-, 8-, or 16-bit correction) or 1024 byte (24-bit correction) error correction code (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 direct memory access (DMA)
- Provides programmable access timing

NAND Flash Controller Block Diagram and System Integration

Figure 10–1 shows integration of the NAND flash controller in the HPS. The flash controller receives commands and data from the host through the command and data slave interface. The host accesses the flash controller's control and status registers (CSRs) through the register slave interface. The flash controller handles all command sequencing and flash device interactions. The bootstrap interface supports configuration of the NAND flash controller when booting the HPS from NAND flash memory. The flash controller generates interrupts to the HPS Cortex™-A9 MPCore™ processor generic interrupt controller. The DMA master interface provides accesses to and from the flash controller through the controller's built-in DMA.

Figure 10–1. NAND Flash Controller Block Diagram



Functional Description of the NAND Flash Controller

This section describes the functionality of the NAND flash controller.

Discovery and Initialization

The NAND flash controller requires 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 Electronic Signature` 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 reset is de-asserted.

To support booting and initialization, the `rdy_busy_in` pin must be connected. The NAND flash controller sends the `reset` command to the connected device.

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 `ONFI Read ID` command to read the ONFI signature from the device, to determine whether an ONFI or a legacy device is connected.
3. If the data returned by the 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 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 device, which is the expected location of the preloader software.

   
   The system manager can bypass this step by asserting `bootstrap_inhibit_b0p0_load` before reset is deasserted.

7. The flash controller sends the `reset` command to the device.
8. The flash controller sets the value of the `rst_comp` bit in the `intr_status0` register in the `status` group.

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 allow 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.

For more information about the bootstrap interface control bits, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Table 10–1 lists the relevant bootstrap setting bits, found in the system manager's `bootstrap` register, in the `nandgrp` group. This table also lists recommended bootstrap settings for a 512 byte page device.

Table 10–1. Recommended Bootstrap Settings for 512 Byte Page Device

Register	Value
`noinit`	1 (1)
`page512`	1
`noloadb0p0`	1
`tworowaddr`	■ 1—flash device supports two-cycle addressing ■ 0—flash device support three-cycle addressing

Note to Table 10–1:

(1) 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".

Configuration by Host

If the system manager sets `bootstrap_inhibit_init` to 1, the NAND flash controller does not perform the discovery and initialization process. 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.

Table 10–2 shows the recommended configuration by host settings to enable the basic read, write, and erase operations for a single-plane, 512 bytes/page device.

Table 10–2. Recommended Bootstrap Settings for 512 Byte Page Device

Register (1)	Value
`devices_connected`	1
`device_width`	0 indicating an 8-bit NAND flash device
`number_of_planes`	1 indicating a single-plane device
`device_main_area_size`	The value of this register must reflect the flash device's page main area size.
`device_spare_area_size`	The value of this register must reflect the flash device's page spare area size.
`pages_per_block`	The value of this register must reflect number of pages per block in the flash device.

Note to Table 10–2:

(1) All registers are in the `config` group.

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.

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`).

Clocks

Table 10–3 lists the NAND flash controller clock inputs.

Table 10–3. Clock Inputs to NAND Flash Controller

Clock Signal	Description
`nand_x_clk`	Clock for master and slave interfaces and the ECC sector buffer
`nand_clk`	Clock for the NAND flash controller

The frequency of `nand_x_clk` is four times the frequency of `nand_clk`.

For more information about the clock inputs, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Resets

The NAND flash controller has one reset signal. The reset manager drives this signal to the NAND flash controller on a cold or warm reset.

Before the NAND flash controller comes out of the reset state, the pin multiplexers for the flash external interface must be configured.

For more information about the reset manager, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Indexed Addressing

The NAND flash controller uses indexed addressing to reduce the address span consumed by the flash controller. Indirect addressing is controlled by two registers, accessed through the command and data slave interface in the `nanddata` map, as described in Table 10–4.

Table 10–4. Register Map for Indexed Addressing

Register Name	Offset Address	Usage
`Control`	0x0	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 Table 10–5 through Table 10–8.
`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 host uses indexed addressing as follows:

1. Program the 32-bit index-address field into the `Control` register at offset 0x0 of the data/command slave port. This action provides the flash address parameters to the NAND flash controller.
2. Perform 32-bit read or write in the `Data` register at offset 0x10 of the data/command slave port.
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.

Table 10–5 lists the address bits as interpreted by the NAND flash controller for a MAP00 command.

Table 10–5. MAP00 Address Mapping

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

The usage of this command under normal operations is limited to the following situations:

- It 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, this method is not recommended 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 NAND flash controller supports 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. You must use the same address until the entire page is transferred, even if multiple commands are required.

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.

Table 10–6 lists the address bits as interpreted by the NAND flash controller for a MAP01 command.

Table 10–6. MAP01 Address Mapping

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*> (1)	BLK_ADDR	Block address in the device
(<*M*>-1):0 (1)	PAGE_ADDR	Page address in the device

Note to Table 10–6:

(1) <*M*> depends on the number of pages per block in the device. <*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

The NAND flash controller incorporates ECC on-the-fly correction that corrects data read from the device internally before transferring the data out from the flash controller. The ECC sector buffers store data, while the ECC engine computes the error location.

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 of 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 `unsup_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.

Table 10–7 lists the address bits as interpreted by the NAND flash controller for a MAP10 command.

Table 10–7. MAP10 Address Mapping

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*> (1)	BLK_ADDR	Block address in the device
(<*M*>-1):0 (1)	PAGE_ADDR	Page address in the device

Note to Table 10–7:

(1) <*M*> depends on the number of pages per block in the device. <*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

Table 10–8 lists the special functions defined by MAP10 command.

Table 10–8. 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

Use the MAP10 command as follows:

- MAP10 commands should be used to issue commands to the device, 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 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 supported by 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/write cycles directly on the NAND device interface. Command, address, and write data values are placed in the `Data` register. On a read, the returned data also appears in the `Data` register. The indirect address register encodes the control operation type. 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. The host can issue only single-beat accesses to the data slave port while using MAP11 commands.

Table 10–9 lists the address bits as interpreted by the NAND flash controller for a MAP11 command.

Table 10–9. MAP11 Addressing Mapping

Address Bits	Name	Description
31:28	(reserved)	Set to 0
27:26	CMD_MAP	Set to 3
25:2	(reserved)	Set to 0
1:0	TYPE	Sets the control type as follows: ■ 0 = Command cycle ■ 1 = Address cycle ■ 2 = Data Read/Write Cycle

Use the MAP11 command 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.

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 data/command slave interface.

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.

When the host issues a data DMA command the NAND flash controller transfers data between the flash device and host memory if data DMA is enabled (the flag bit of the dma_enable register in the dma group is set to 1). On the completion of the transfer the flash controller informs the host by asserting an interrupt.

- 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 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.

Multitransaction DMA Command

To initiate DMA with a multitransaction DMA command, you send four command-data pairs to the NAND flash controller's data and control slave port, as shown in Table 10–10 through Table 10–13.

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.

Table 10–10 through Table 10–13 show the format of each command-data pair.

Table 10–10. Command-Data Pair 1

	31:28	27:26	25:24	23:<*M*> (1)	(<*M*> – 1):0 (1)
Command	0x0	0x2	0x0	Block address	Page address

	31:16	15:12	11:8	7:0
Data	0x0	0x2	0x0 = Read 0x1 = Write	<*PP*>= Number of pages

Note to Table 10–10:

(1) <*M*> depends on the number of pages per block in the device. <*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

Table 10–11. Command-Data Pair 2

	31:28	27:26	25:24	23:8	7:0
Command	0x0	0x2	0x0	memory address high (1)	0x0

	31:16	15:12	11:8	7:0
Data	0x0	0x2	0x2	0x0

Note to Table 10–11:

(1) The buffer address in host memory, which must be aligned to 32 bits

Table 10–12. Command-Data Pair 3

	31:28	27:26	25:24	23:8	7:0
Command	0x0	0x2	0x0	memory address low (1)	0x0

	31:16	15:12	11:8	7:0
Data	0x0	0x2	0x3	0x0

Note to Table 10–12:

(1) The buffer address in host memory, which must be aligned to 32 bits

Table 10–13. Command-Data Pair 4

	31:28	27:26	25:24	23:17	16	15:8	7:0
Command	0x0	0x2	0x0	0x0	INT (1)	Burst length	0x0

	31:16	15:12	11:8	7:0
Data	0x0	0x2	0x4	0x0

Note to Table 10–13:

(1) INT specifies the host interrupt to be generated at the end of the complete DMA transfer. 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.

For more information, refer to "Indexed Addressing" on page 10–6.

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.

For more information about the system manager, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 Table 10–10 through Table 10–13 before sending a different command.

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").

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, in the format shown in Table 10–14, as an INCR4 command. The address decoding for MAP10 pipeline DMA command remains the same, as shown in Table 10–7 on page 10–9. Table 10–14 lists the MAP10 burst DMA command structure. The burst DMA command carries the same information as the multitransaction DMA command-data pairs, but in a very different format.

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 "Multitransaction DMA Command").

Table 10–14. MAP10 Burst DMA (INCR4) Command Structure

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 (1)	memory address high															
Beat 2 (1)	memory address low															
Beat 3	0x0							INT (2)	Burst length							

Notes to Table 10–14:

(1) The buffer address in host memory, which must be aligned to 32 bits

(2) INT specifies the host interrupt to be generated at the end of the complete DMA transfer. 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.

You can optionally send the 16-bit fields in Table 10–14 to the NAND flash controller as four separate bursts of length 1 in sequential order. Altera recommends this method.

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 modify its cache capabilities.

For more information about the system manager, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 the presence of errors is calculated according to 512 or 1024 bytes of data read.

Table 10–15 lists the relationship between different correction capabilities, sector sizes, and the required check-bit sizes written into the spare area.

Table 10–15. 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	46

The NAND flash controller provides the following ECC programming modes that software uses to format a page:

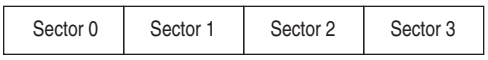
- Main Area Transfer Mode
- Spare Area Transfer Mode
- Main+Spare Area Transfer Mode

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 10–2 shows the main area transfer mode programming model.

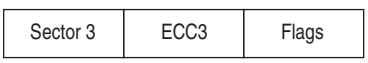
Figure 10–2. Main Area Transfer Mode Programming Model for ECC



Spare Area Transfer Mode

The NAND flash controller does not introduce or interpret ECC check bits in spare area transfer mode, and acts as pass through for data transfer. Figure 10–3 shows the spare area transfer mode programming model.

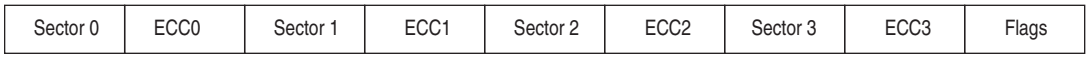
Figure 10–3. Spare Area Transfer Mode Programming Model for ECC



Main+Spare Area Transfer Mode

In main+spare area transfer mode, the NAND flash controller expects software to format a page as shown in Figure 10–4. 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 10–4. Main+Spare Area Transfer Mode Programming Model for ECC



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.

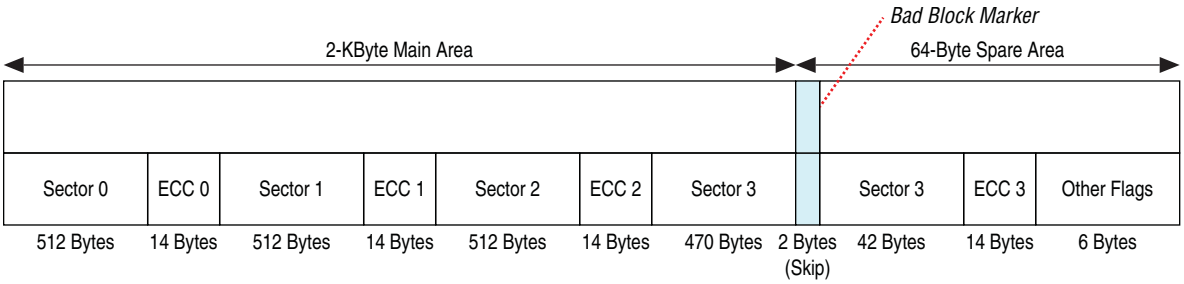
In the spare area transfer mode, the NAND flash controller ignores the `spare_area_skip_bytes` and `spare_area_marker` registers. The flash controller transfers the data exactly as received from the host or device.

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.

For more information about the formatting of this data, refer to Figure 10–4 on page 10–17.

Figure 10–5 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.

Figure 10–5. Bad Block Marker



For detailed information about configuring the NAND flash controller for default, spare, or main+spare area transfer mode, refer to "Transfer Mode Operations" on page 10–25.

Error Correction Status

The ECC error correction information (`ECCCorInfo_b01`) register, in the `ecc` group, contains error correction information for each read or write that the NAND flash controller performs. The `ECCCorInfo_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, `ECCCorInfo_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 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.

Interface Signals

Table 10–16 lists I/O pin use for the NAND flash interface signals.

Table 10–16. 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

The HPS I/O pins support a single x8 device.

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).

If you write a configuration register and follow it up with a data operation that is dependent on the value of this configuration register, Altera 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 software to access and control 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_cyccnt` 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 needs to 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, 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" on page 10–2.

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.

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 may choose not to use these sequences and 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, then 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 need to decide whether you want to have the ECC enabled or disabled. If the ECC needs to be enabled, then 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 need to 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.

For detailed information about ECCs, refer to "ECC" on page 10–15.

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 chooses to use the DMA logic in the flash controller, then the appropriate DMA enable and interrupts bits in the register space must be set.

- Set the `flag` bit in the `global_int_enable` register in the `config` group to 1, to enable global interrupt.
- Set the relevant bits of the `intr_en0` register in the `status` group to 1 before sending any operations if the flash controller is in interrupt mode.
- Enable DMA if your application needs DMA mode. Enable DMA by setting the `flag` bit of the `dma_enable` register in the `dma` group. Altera 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.
- 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 interrupt status bit is the last to be asserted during a DMA operation to transfer data. This bit signifies the completion of data transfer sequence.
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)
   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. The data transfer mode can be initialized using MAP10 commands.
- The `write_protect` register need not be initialized 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" on page 10–6.

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 Table 10–7 on page 10–9.

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 device completes 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.

Refer to"MAP10 Commands" on page 10–9 for detailed information about the MAP10 commands. Table 10–17 lists the functionality of the MAP10 transfer mode commands, and their mappings to the transfer_spare_reg register in the config group.

Table 10–17. transfer_spare_reg and MAP10 Transfer Mode Commands

transfer_spare_reg	MAP10 Transfer Mode Commands	Resulting NAND Flash Controller Mode
0	0x42	Main (1)
0	0x41	Spare
0	0x43	Main+spare
1	0x42	Main+spare (1)
1	0x41	Spare
1	0x43	Main+spare

Note to Table 10–17:

(1) Default access mode (0x42) maps to either main (only) or main+spare mode, depending on the value of transfer_spare_reg.

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.

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.

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. The flow starts 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. Next provide 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 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. Altera 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.

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.

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.

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.

Pipeline read-ahead commands can read data from the queue in this interleaved fashion. The parameter <*PP*> 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.

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.

Since 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 happens immediately.

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.

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 case of pipeline read command and start of data transfer of the last page to be programmed in case of pipeline writes command.

An additional `program_comp` interrupt is generated when the last page program operation completes in case of 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.

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 address is updated in the `err_block_addr0` and `err_page_addr0` registers in the `status` group.

Pipeline commands sequence advanced commands in the device like 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.

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 <*PP*> 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 must be issued 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.

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 actually 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.

NAND Flash Controller Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the following links for the module instance:

- **nanddata**
- **nandregs**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 10–18 lists the revision history for this document.

Table 10–18. Document Revision History

Date	Version	Changes
November 2012	1.2	■ Supports one 8-bit device ■ Show additional supported block sizes ■ Bad block marker handling
May 2012	1.1	Added programming model section.
January 2012	1.0	Initial release.

11. SD/MMC Controller

cv_54011-1.1

The hard processor system (HPS) provides a Secure Digital/MultiMediaCard (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) nonremovable flash devices.

The SD/MMC controller is based on the Synopsys® DesignWare® Mobile Storage Host (DWC_mobile_storage) 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* as described in "References" on page 11–79.

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:
  - SD, including eSD—version 3.0
  - SDIO, including embedded SDIO (eSDIO)—version 3.0
  - CE-ATA—version 1.1
  - MMC, including eMMC—version 4.41, 1-bit, 4-bit, and 8-bit (in some packages, as described in Table 11–2 on page 11–3)
- Integrated descriptor-based direct memory access (DMA)
- Internal 4 KB receive and transmit FIFO buffer

Table 11–1 shows various SD card device types and the supported voltages, bus modes, and speeds.

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).

Table 11–1. SD Card Use Cases

Card Device Type	Voltages Supported		Bus Modes Supported				Bus Speed Modes Supported			
							Default Speed	High Speed	SDR12	SDR25 (1)
	3.3 V	1.8 V	SPI	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	✓	✓ (2)	✓	✓	✓	—	✓	✓	✓	✓
SDXC	✓	✓ (2)	✓	✓	✓	—	✓	✓	✓	✓
eSD	✓	✓ (2)	✓	✓	✓	—	✓	✓	✓	✓
SDIO	✓	✓ (2)	✓	✓	✓	—	✓	✓	✓	✓
eSDIO	✓	✓ (2)	✓	✓	✓	✓ (3)	✓	✓	✓	✓

Notes to Table 11–1:

(1) 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).

(2) Controls the voltage switch output to support 1.8-V signalling for SD.

(3) Optional 8-bit bus mode for eSDIO is not supported in all FPGA packages.

Card form factors (such as mini and micro) are not enumerated in Table 11–1 because they do not impact the card interface functionality.

Table 11–2 shows various MMC card device types and the supported voltage, bus modes, and bus speeds.

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.

Table 11–2. MMC Use Cases

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	SPI (1)	1 bit	4 bit	8 bit	Default Speed	High Speed
MMC	20	2.5	✓	—	✓	✓	—	—	✓	—
RSMMC	20	10	✓	—	✓	✓	✓	—	✓	✓
MMCPlus	50 (3)	25	✓	—	—	✓	✓	✓ (2)	✓	✓
MMCMobile	50	6.5	✓	✓	—	✓	—	—	✓	✓
eMMC	50	25	✓	✓	—	✓	✓	—	✓	✓

Notes to Table 11–2:

(1) SPI mode is obsolete in the MMC 4.41 specification.

(2) The optional 8-bit bus mode is not supported in all FPGA packages.

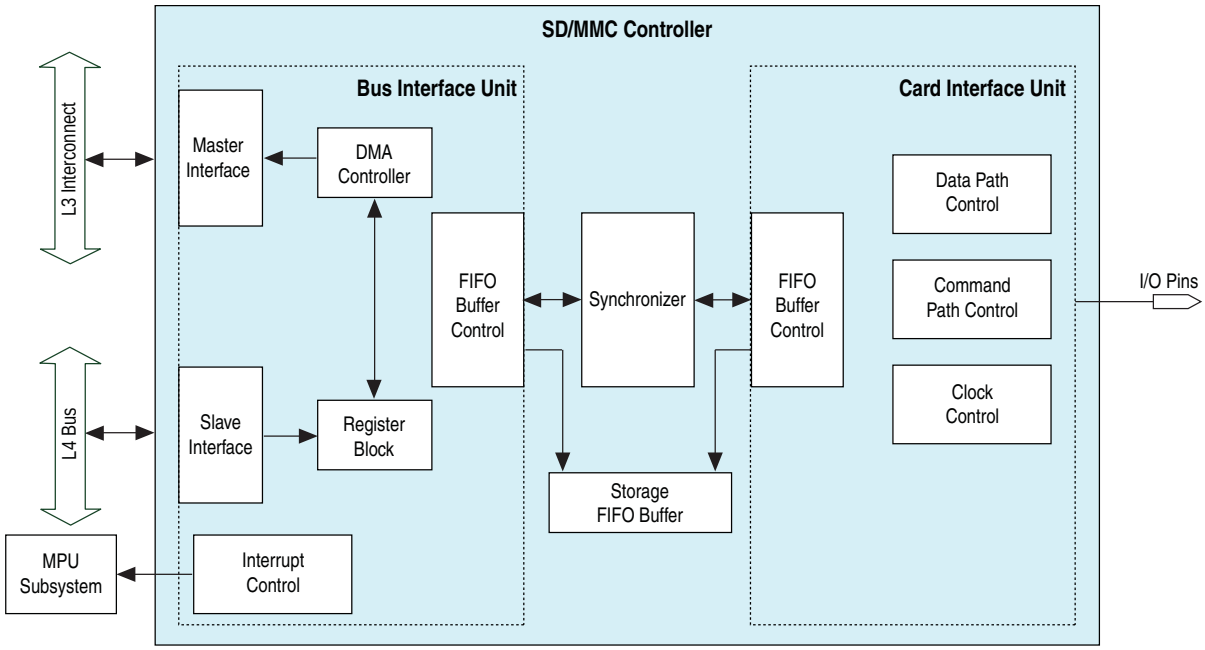
(3) Supports a maximum clock rate of 50 MHz instead of 52 MHz (specified in MMC specification).

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.

Figure 11–1 shows a block diagram of the SD/MMC controller and how it integrates in the HPS.

Figure 11–1. SD/MMC Controller Connectivity



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.

Figure 11–2 illustrates an example of a multiple-block read operation. The clock is a representative only and does not show the exact number of clock cycles.

Figure 11–2. Multiple-Block Read Operation

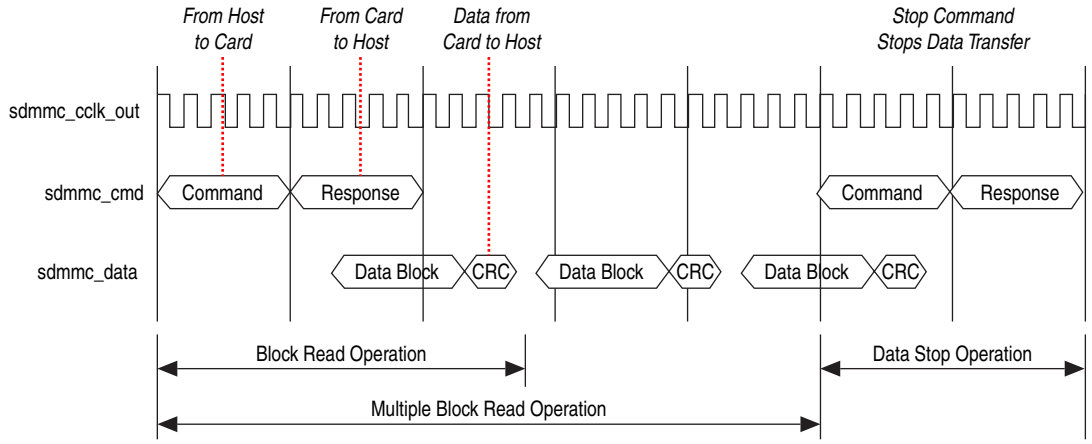
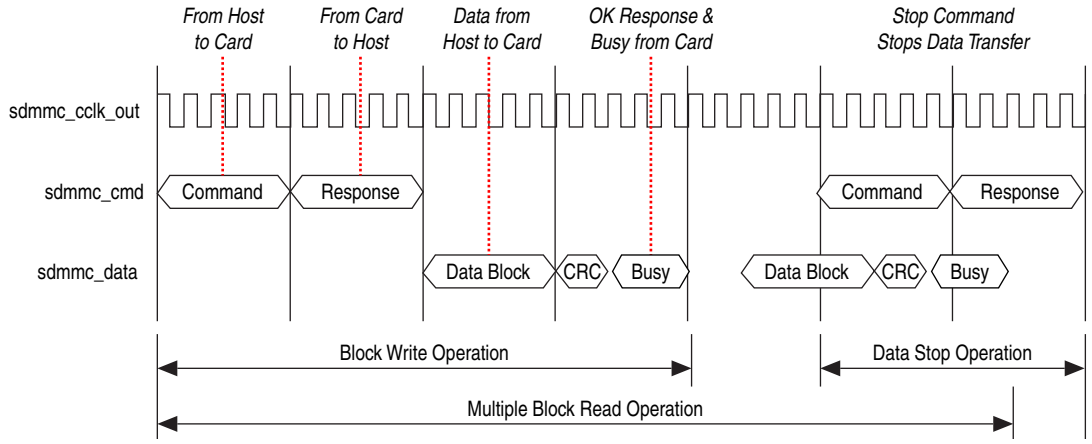


Figure 11–3 illustrates an example of a command token sent by the host in a multiple-block write operation.

Figure 11–3. Multiple-Block Write Operation



BIU

The BIU interfaces with the 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:

- Slave interface
- Register block
- FIFO buffer
- 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. 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 transferred 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.

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 once 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.

After 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 to 1 until the software resets the bit to 0 by writing a 1 to the interrupt bit; writing a 0 leaves the bit untouched.

The interrupt port is an active-high, level-sensitive interrupt. The interrupt port is active 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 following bits are available as top-level ports for debug purposes:

- All bits of the `intmask` register
- All bits of the `rintsts` register
- The `int_enable` bit of the `ctrl` register

The `int_enable` bit of the `ctrl` register is set to 0 on power-on, and the `intmask` register bits are set to 0x0000000, which masks all the interrupts.

The following conditions can cause the interrupt to occur:

- End-bit error on read
- No cyclic redundancy code (CRC) on write
- Auto command done
- Start-bit error
- Hardware locked write error
- FIFO buffer underflow or overflow error
- Data starvation by host timeout
- Data read timeout or boot data start
- Response timeout or boot ACK received
- Data CRC error
- Response CRC error

- Receive FIFO buffer data request
- Transmit FIFO buffer data request
- Data transfer over
- Command done
- Response error

The 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.

For more information, refer to the *System Manager* chapter in volume 3 of the *Cyclone® V Device Handbook*.

Internal DMA Controller

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 set up 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`).
- Chain structure—Each descriptor points to a unique buffer, and to the next descriptor in a linked list.

Figure 11–4 and Figure 11–5 illustrate the internal DMA controller dual-buffer descriptor structure and chain descriptor structure respectively.

Figure 11–4. Dual-Buffer Descriptor Structure

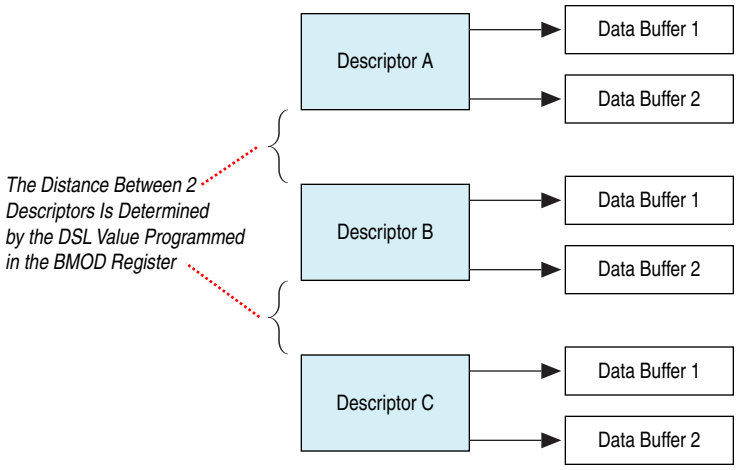


Figure 11–5. Chain Descriptor Structure

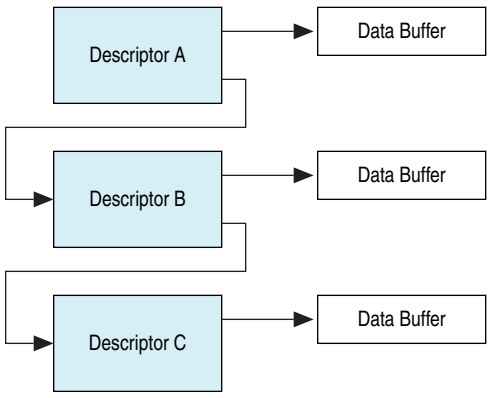


Table 11–3 shows the internal format of a descriptor. The descriptor address must be aligned to the 32-bit bus. Each descriptor contains 16 bytes of control and status information. For information about each of the bits of the descriptor, refer to Table 11–4 on page 11–11 through Table 11–7 on page 11–12.

Table 11–3. Descriptor Format

Name	Offset	31	30	29 ... 26	25 ... 13	12 ... 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										

The DES0 field in the internal DMA controller descriptor contains control and status information. Table 11–4 lists the bits in this descriptor.

Table 11–4. 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 `rintsts` register. ■ End-bit error (`ebe`) ■ Response timeout (`rto`) ■ Response CRC (`rcrc`) ■ Start-bit error (`sbe`) ■ Data read timeout (`drto`) ■ Data CRC for receive (`dcrc`) ■ Response error (`re`)
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 (DES1[25:13]) must be all zeros.
3	First Descriptor (FS)	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 (`idsts`) for the data that ends in the buffer pointed to by this descriptor.
0	Reserved	—

The DES1 descriptor field contains the buffer size. Table 11–5 lists the bits in this descriptor.

Table 11–5. Internal DMA Controller DES1 Descriptor Field

Bits	Name	Description
31:26	Reserved	—
25:13	Buffer 2 Size (BS2)	These bits indicate 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.

The DES2 descriptor field contains the address pointer to the data buffer. Table 11–6 lists the bits in this descriptor.

Table 11–6. Internal DMA Controller DES2 Descriptor Field

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.

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. Table 11–7 lists the bits in this descriptor.

Table 11–7. Internal DMA Controller DES3 Descriptor Field

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 execute 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 (`pbl`) 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 INCR4/8/16 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 FSM

The following steps show the internal DMA controller functional state machine (FSM) operations:

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.

1. 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.
2. 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.

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.

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.

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.

Table 11–8 lists the legal PBL and FIFO buffer watermark values for internal DMA controller data transfers operations.

Table 11–8. PBL and Watermark Levels

PBL (Number of transfers)	TX/RX 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

Fatal Bus Error Scenarios

A fatal bus error occurs due to an error response through the master interface. 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.

CIU

The 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_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.

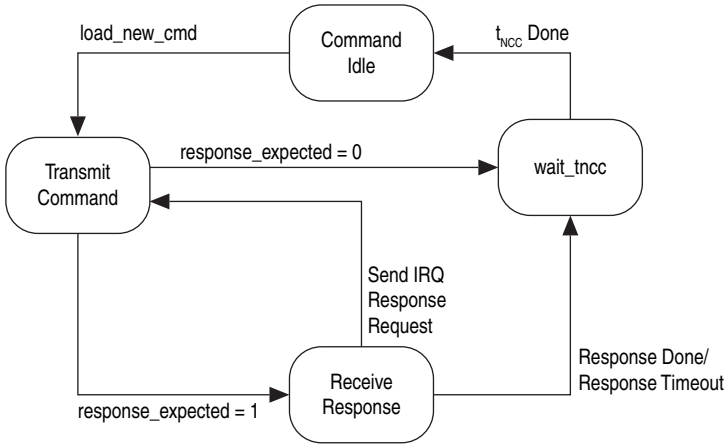
Loading a new command from the BIU in the command path depends on the following `cmd` register bit settings:

- `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 11–6 illustrates the command path state machine.

Figure 11–6. 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 `tmout` register), the `rto` bit and command done (`CD`) bit are set to 1 in the `rintsts` 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 `rintsts` 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 `rcrc` bit is set to 1 in the `rintsts` 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 `rintsts` 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 `rintsts` register.

A correct card response contains the fields listed in Table 11–9. The command path verifies the contents of the card response.

Table 11–9. 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.

For more information about response values, refer to *Physical Layer Simplified Specification, Version 3.01* as described in "References" on page 11–79.

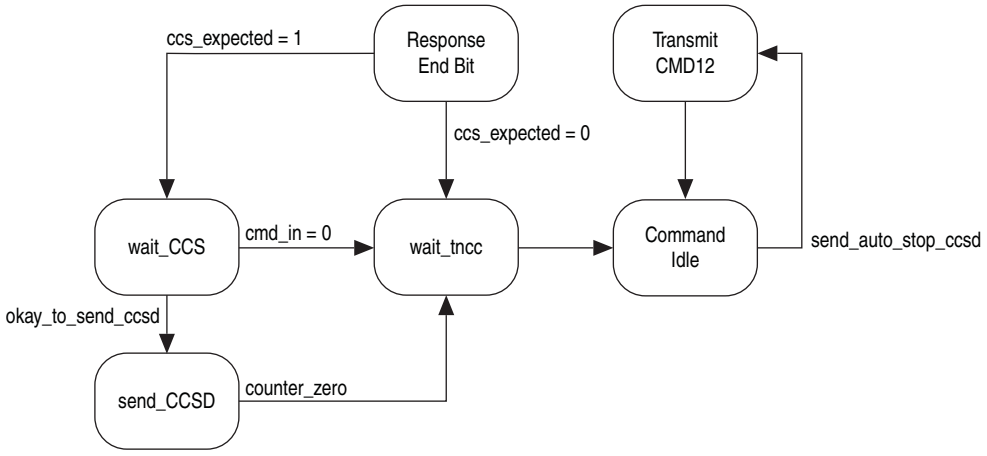
Driving P-bit on CMD Line

The command path drives a one-cycle pull-up bit (P-bit) to 1 on the CMD line 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 Figure 11–7 according to `cmd` register bit values.

Figure 11–7. CE-ATA Command Path State Machine



The following describe some of the details in Figure 11–7:

1. 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.
2. 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:
   a. 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.
   b. Receive the CCS on the CMD line.
3. 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.

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.

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.

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 (DRTO) 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

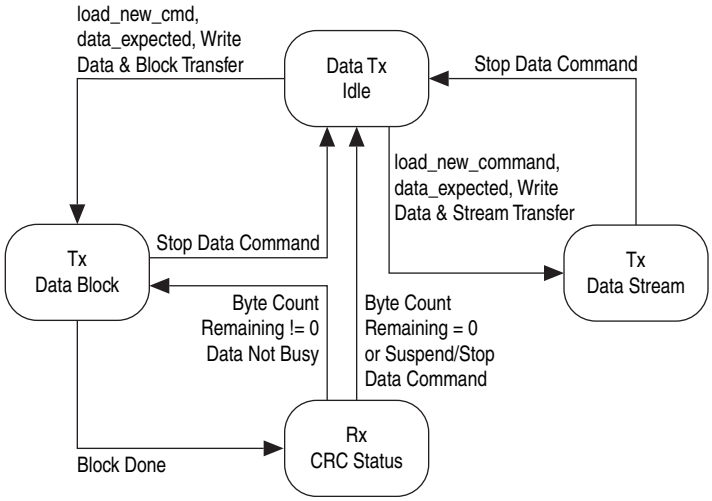
The data path block reads the data FIFO buffer and transmits data on the card bus during a write data transfer, or receives data and writes it to the FIFO buffer during a read data transfer. The data path loads new data parameters—data expected, read/write data transfer, stream/block transfer, block 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, illustrated in Figure 11–8, 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 11–8. 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.

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.

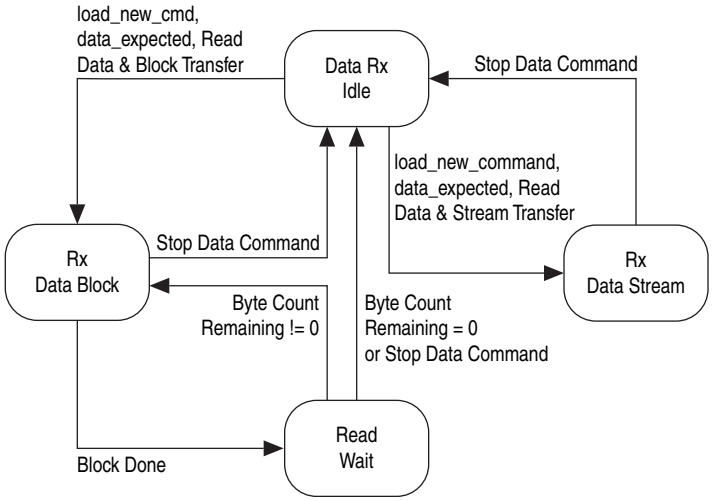
If the bytcnt is zero (the block size must be greater than zero) the transfer is an open-ended block transfer. The data transmit state machine for this type of data transfer continues the block-write data transfer until the host software issues an SD/SDIO STOP or STOP_TRANSMISSION (CMD12) command.

Data Receive

The data-receive state machine, illustrated in Figure 11–9, 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 11–9. 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 set to 0 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 details listed in Table 11–10.

Table 11–10. Auto-Stop Generation

ırd type	Transfer type	Byte Count	send_auto_stop bit set	Comments
MC	Stream read	0	No	Open-ended stream
MC	Stream read	> 0	Yes	Auto-stop after all bytes transfer
MC	Stream write	0	No	Open-ended stream
MC	Stream write	> 0	Yes	Auto-stop after all bytes transfer
MC	Single-block read	> 0	No	Byte count = 0 is illegal
MC	Single-block write	> 0	No	Byte count = 0 is illegal
MC	Multiple-block read	0	No	Open-ended multiple block
MC	Multiple-block read	> 0	Yes **	Pre-defined multiple block
MC	Multiple-block write	0	No	Open-ended multiple block
MC	Multiple-block write	> 0	Yes **	Pre-defined multiple block
)MEM	Single-block read	> 0	No	Byte count = 0 is illegal
)MEM	Single-block write	> 0	No	Byte count = 0 illegal
)MEM	Multiple-block read	0	No	Open-ended multiple block
)MEM	Multiple-block read	> 0	Yes	Auto-stop after all bytes transfer
)MEM	Multiple-block write	0	No	Open-ended multiple block
)MEM	Multiple-block write	> 0	Yes	Auto-stop after all bytes transfer
SDIO	Single-block read	> 0	No	Byte count = 0 is illegal
SDIO	Single-block write	> 0	No	Byte count = 0 illegal
SDIO	Multiple-block read	0	No	Open-ended multiple block
SDIO	Multiple-block read	> 0	No	Pre-defined multiple block
SDIO	Multiple-block write	0	No	Open-ended multiple block
SDIO	Multiple-block write	> 0	No	Per-defined multiple block

** 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 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.

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.

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. Table 11–11 lists the commands and their register setup requirements.

Table 11–11. 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)
cmd Register Setup						
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
cmdarg Register Setup						
	Stuff bits	32-bit write protect data address	Stuff bits	Stuff bits	Stuff bits	Stuff bits
blksiz Register Setup						
	16	4	Num_bytes (1)	64	4	8
bytcnt Register Setup						
	16	4	Num_bytes (1)	64	4	8

Note to Table 11–11:

(1) Num_bytes = Number of bytes specified as per the lock card data structure. Refer to the SD specification and the MMC specification.

Table 11–12.

	CMD27	CMD30	CMD42	ACMD13	ACMD22	ACMD51
Command register programming						
Cmd_index	6'h1B	6'h1E	6'h2A	6'h0D	6'h16	6'h33
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
Command Argument register programming						
	Stuff bits	32-bit write protect data address	Stuff bits	Stuff bits	Stuff bits	Stuff bits
Block Size register programming						
	16	4	Num_bytes*	64	4	8
Byte Count register programming						
	16	4	Num_bytes*	64	4	8
*: Num_bytes = No. of bytes specified as per the lock card data structure (Refer to the SD specification and the MMC specification).						

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:

- `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 cclk_out card output clock can be enabled or disabled under the following conditions:
  - cclk_out is enabled when the cclk_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 cclk_low_power bit of the clkena register to 1. If low-power mode is enabled to save card power, the cclk_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.

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

The SD/MMC controller clocks are listed in Table 11–13.

Table 11–13. SD/MMC Controller Clocks (Part 1 of 2)

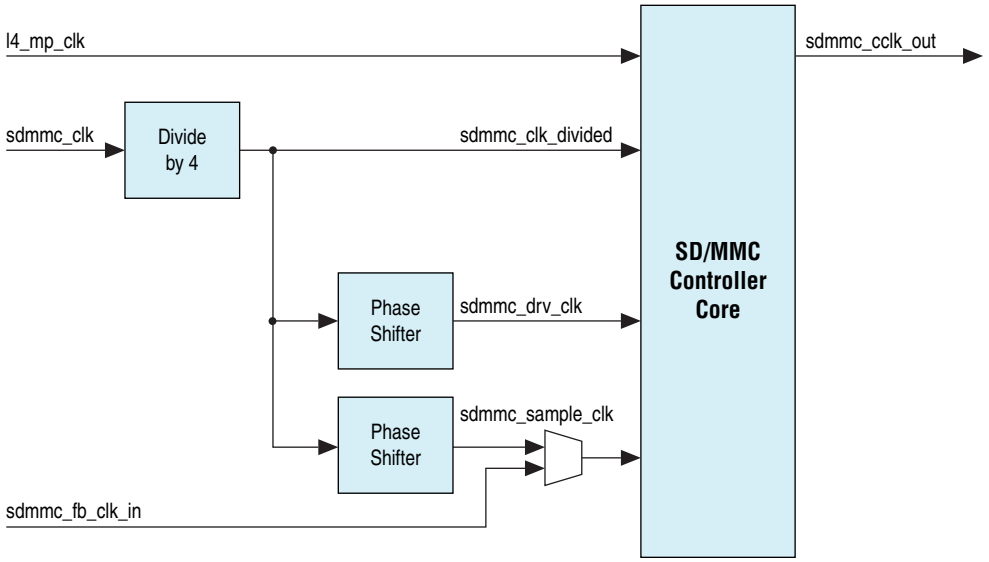
Clock Name	Direction	Description
`sdmmc_clk`	In	Clock for SD/MMC controller CIU
`l4_mp_clk`	In	Clock for SD/MMC controller BIU
`sdmmc_cclk_out`	Out	Generated output clock for card
`sdmmc_sample_clk`	Internal	Phase-shifted clock of `sdmmc_clk` used to sample the command and data from the card

Table 11–13. SD/MMC Controller Clocks (Part 2 of 2)

Clock Name	Direction	Description
sdmmc_drv_clk	Internal	Phase-shifted clock of sdmmc_clk for controller to drive command and data to the card to meet hold time requirements
sdmmc_clk_divided	Internal	Divide-by-four clock of sdmmc_clk

Figure 11–10 shows the connection of various clocks to the controller.

Figure 11–10. SD/MMC Controller Clock Connections



The sdmmc_clk clock from the clock manager is divided by four (sdmmc_clk_divided clock) before passed to the phase shifters and controller. 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. 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 "Clock Setup" on page 11–40.

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 "Clock Control Block" on page 11–31.

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.

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Interface Signals

Table 11–14 lists I/O pin use of the SD/MMC controller interface signals.

Table 11–14. SD/MMC Controller Interface I/O Pins

Signal	Width	Direction	Description
sdmmc_cclk_out	1	Out	Clock from controller to the card
sdmmc_cmd	1	In/Out	Card command
sdmmc_pwren	1	Out	External device power enable
sdmmc_data	8	In/Out	Card data

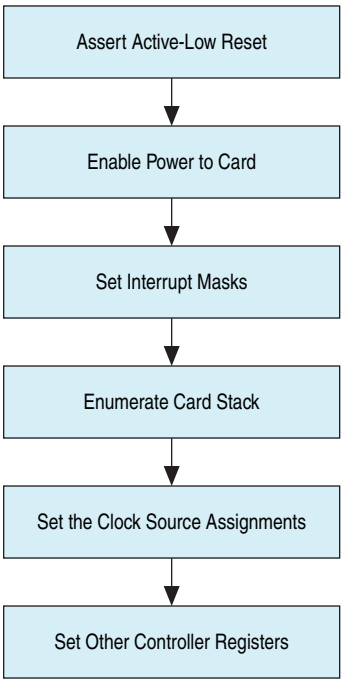
SD/MMC Controller Programming Model

Initialization

This section describes how to initialize the SD/MMC controller.

Figure 11–11 shows the initialization flow of the SD/MMC controller. 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 11–11. SD/MMC Controller Initialization 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.

   Altera 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" on page 11–37.
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" on page 11–40.
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 0x64.
   - The data timeout field (`data_timeout`) of the `tmout` register, highest of the following:
     - $10 * N_{AC}$

       $= 10 * ((TAAC * F_{OP}) + (100 * NSAC))$

       where:

       N_{AC} = card device total access time

       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 (`debnce`). 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.

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. 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. 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.
   a. 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 in "References" on page 11–79:

   b. 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 step e.

   c. 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

   d. If a response is received to the previous SD_SEND_OP_COND command, 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.

   e. If a response is not received to the initial SEND_IF_COND command, the card does not support High Capacity SD2.0. 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

   f. 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.

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.

4. Use the following sequence to determine whether the card is a CE-ATA 1.1, CE-ATA 1.0, or MMC device:
   a. Determine whether the card is a CE-ATA v1.1 card device by attempting to select ATA mode. 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. 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.

        You can verify the currently selected mode by reading it back from byte 191 of the EXT_CSD register.

        Skip to step 5.
      - If the card device does not support ATA mode, it might be an MMC card or a CE-ATA v1.0 card. Continue to Step b.

   b. 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.
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

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, as described in this section. 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 "Interrupt and Error Handling" on page 11–66.
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 (`smplsel`) 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_cclk_out` clock when the card is idle for more than eight clock cycles.

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.

  ☞ Ensure that the DMA is idle before performing a DMA reset. Otherwise, the L3 interconnect might be left in an indeterminate state.

Altera 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 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.

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 Table 11–15 on page 11–44.

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.

☞ Software cannot modify clock parameters while a command is being executed.

Table 11–15. cmd Register Settings for Non-Data Transfer Command

Parameter	Value	Comment
Default		
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 the speed mode being used.
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
User-Selectable		
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. Altera 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.

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.

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.

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.

Conditions 6, 7, and 8 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" on page 11–64.

   ☞ 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 Table 11–16. 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. 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.

Table 11–16. cmd Register Settings for Single-Block and Multiple-Block Reads

Parameter	Value	Comment
Default		
`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 being 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
`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	Set according to Table 11–10 on page 11–28
`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
User-Selectable		
`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 (CMS17) or READ_MULTIPLE_BLOCK (CMD18).

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 Table 11–17 on page 11–49. 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.

Table 11–17. cmd Register Settings for Single-Block and Multiple-Block Write (Part 1 of 2)

Parameter	Value	Comment
Default		
`start_cmd`	1	This bit resets itself to 0 after the command is committed (accepted by the BIU).
`use_hold_reg`	1 or 0	Choose the value based on speed mode being 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
`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	Set according to Table 11–10 on page 11–28
`read_write`	1	Write to card
`response_length`	0	Can be 1 for R2 (long) response
`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 11–17. cmd Register Settings for Single-Block and Multiple-Block Write (Part 2 of 2)

Parameter	Value	Comment
Default		
User-Selectable		
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).

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" on page 11–46, 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" on page 11–48, except for the following bits in the cmd register:

- transfer_mode = 0x1 (for stream transfer)
- cmd_index = 11 (SD/SDIO CMD11)

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 will be 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.

For more information about packed commands, refer to *JEDEC Standard No. 84-A441*, as referenced in "References" on page 11–79.

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. For more information, refer to "Non-Data Transfer Commands" on page 11–42.

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 0x0 – 0xFF.

The ABORT command is a non-data transfer command. For more information, refer to "Non-Data Transfer Commands" on page 11–42.

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 Table 11–18.
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* as described in "References" on page 11–79.

Table 11–18. 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
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:

1. 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, Altera 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.
2. 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.

3. 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.

   The next two sections, "Internal DMA Controller Transmission Sequences" and "Internal DMA Controller Reception Sequences" on page 11–54, describe this step in detail.

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, 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 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 `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. This can be done through 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.

   ☞ 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.

   ☞ For detailed information about the RF<*n*> flags, refer to *SDIO Simplified Specification Version 2.00* as described in "References" on page 11–79:

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 Table 11–19.

Table 11–19. 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 `blksiz` 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" on page 11–46 and "Single-Block or Multiple-Block Write" on page 11–48.
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.

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. 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. For information about the basic settings and interrupts generated for different conditions, refer to "Data Transfer Commands" on page 11–44.

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.

For information about the MMC TRAN state, MMC reset and initialization, refer to *JEDEC Standard No. 84-A441*, as referenced in "References" on page 11–79.

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.

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.

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 Table 11–20 through Table 11–23.

Table 11–20. 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 11–21. cmd Register Settings for ATA Task File Transfer (Part 1 of 2)

Bit	Value	Comment
`start_cmd`	1	-
`ccs_expected`	0	CCS is not expected
`read_ceata_device`	0 or 1	Set to 1 if RW_BLK or RW_REG read
`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. 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).

Table 11–21. cmd Register Settings for ATA Task File Transfer (Part 2 of 2)

Bit	Value	Comment
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 11–22. blksiz Register Settings for ATA Task File Transfer

Bits	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 11–23. bytcnt Register Settings for ATA Task File Transfer

Bits	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.

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.

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.

You must set the cmdarg, cmd, blksiz, and bytcnt registers according to Table 11–24 through Table 11–27.

Table 11–24. 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 11–25. cmd Register Settings for ATA Payload Transfer

Bits	Value	Comment
start_cmd	1	-
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 11–26. blksiz Register Settings for ATA Payload Transfer

Bits	Value	Comment
31:16	0	Reserved bits set to 0
15:0 (block_size)	512, 1024 or 4096	MMC block size can be 512, 1024 or 4096 bytes as negotiated by host

Table 11–27. bytcnt Register Settings for ATA Payload Transfer

Bits	Value	Comment
31:0	<*n*>*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. (<*n*>*block_size = <*x*>*4 KB, where <*n*> and <*x*> 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 `rintsts` 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" on page 11–51.
- 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 `rintsts` register to 1. The `send_auto_stop_ccsd` bit must be set to 1 along with setting the `send_ccsd` bit.

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" on page 11–61.
- If the CCS is not expected from the CE-ATA card device, perform the following steps:
  a. Send an external STOP command.
  b. Terminate the data transfer between the controller and 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`.

Recommended Usage Guidelines for Card Read Threshold

1. The `cardthrctl` register must be set before setting the `cmd` register for a data read command.
2. The `cardthrctl` register must not be set while a data transfer command is in progress.
3. The `cardrdthreshold` field of the `cardthrctl` 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.

☞ If the `cardrdthreshold` field of the `cardthrctl` 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:

   (block size / 4) ≤ 1024
3. Choose the card read threshold value:
   - If (block size / 4) ≥ 512, choose `cardrdthreshold` such that `cardrdthreshold` ≤ (block size / 4) in bytes
   - If (block size / 4) < 512, choose `cardrdthreshold` such that `cardrdthreshold` = (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 (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 Table 11–28.

Table 11–28. Legal Values of dw_dma_multiple_transaction_size and rx_wmark for Block Size = 512

Block Size	dw_dma_multiple_transaction_size	rx_wmark
512	1	0
512	4	3
512	8	7
512	16	15
512	32	31
512	64	63
512	128	127

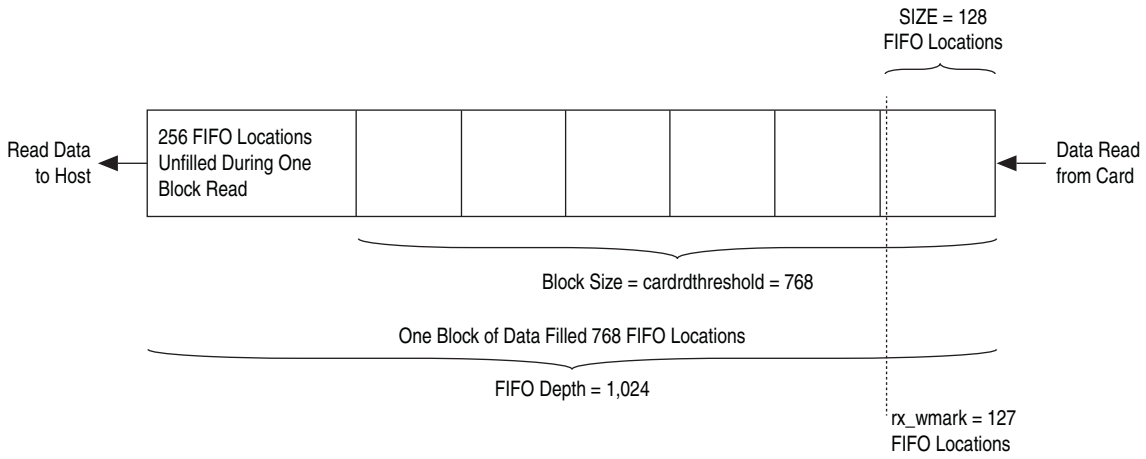
Card Read Threshold Programming Examples

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 block size%size = 0.

- Set the rx_wmark field = size – 1. For example, the rx_wmark field = 128 – 1 = 127.
- Because block size > ½ FifoDepth, set the cardrdthreshold field to the block size. For example, the cardrdthreshold field = 3072 bytes.

Figure 11–12. 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 tcbcnt 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/overrun error—If the FIFO buffer is full and software tries to write data to the FIFO buffer, an overrun 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.
  - 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.

    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.

Booting Operation for eMMC and MMC

This section describes how to set up the controller for eMMC and MMC boot operation.

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.

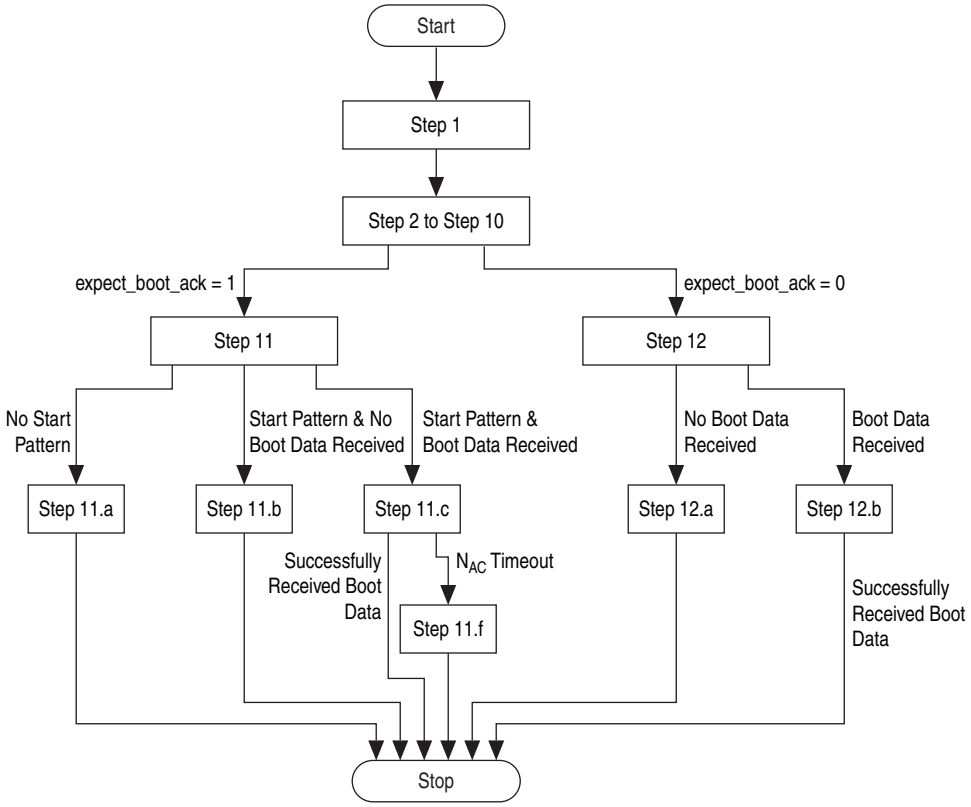
For information about this boot method, refer to the following specifications, as referenced in "References" on page 11–79:

- *JEDEC Standard No. 84-A441*
- *JEDEC Standard No. 84-A44*
- *JEDEC Standard No. JESD84-A43*

Boot Operation for eMMC Card Device

Figure 11–13 illustrates the steps to perform the boot process for eMMC card devices. The detailed steps are described following the flow chart.

Figure 11–13. 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.

   ☞ Altera 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" on page 11–53 and "Internal DMA Controller Reception Sequences" on page 11–54.
   - 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" on page 11–40.
5. Set the `data_timeout` field of the `tmout` 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.

Boot Operation for Removable MMC4.3, MMC4.4 and MMC4.41 Cards

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

For these cards, the software driver must perform the following steps:

1. Discover the card as described in "Enumerated Card Stack" on page 11–37.
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.

   For more information, refer to "Access to Boot Partition" in the following specifications, as referenced in "References" on page 11–79:

   - *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" on page 11–74.

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.

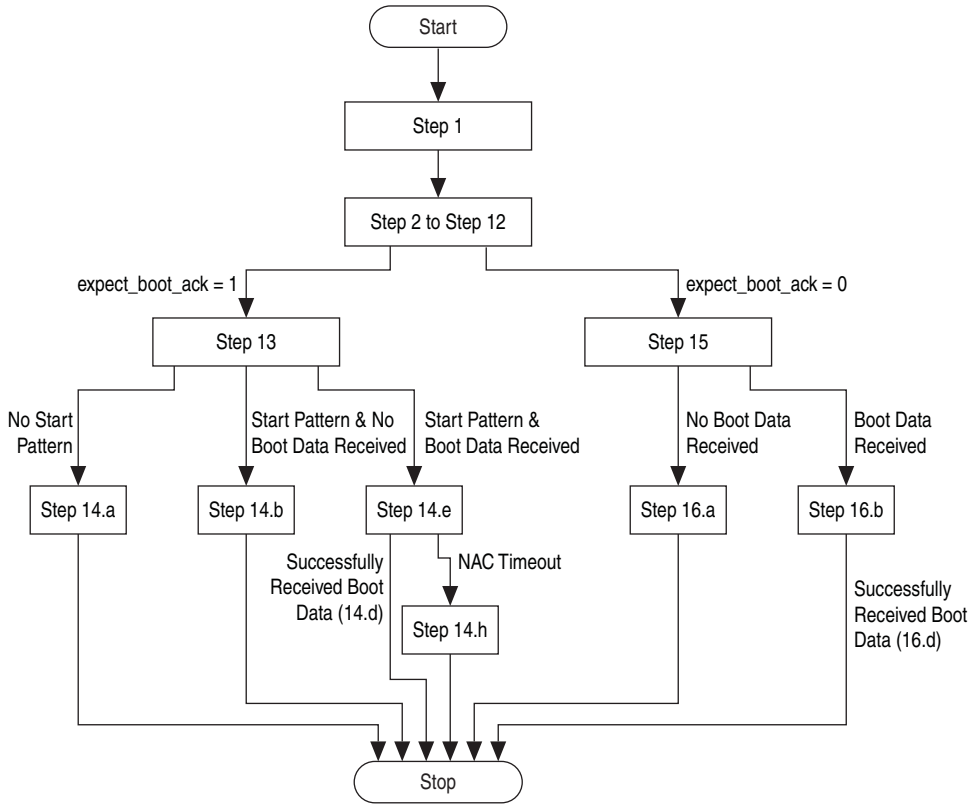
For detailed information about alternative boot operation, refer to the following specifications, as referenced in "References" on page 11–79:

- *JEDEC Standard No. 84-A441*
- *JEDEC Standard No. 84-A44*
- *JEDEC Standard No. JESD84-A43*

Alternative Boot Operation for eMMC Card Devices

Figure 11–14 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 11–14. 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.

☞ Altera 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" on page 11–53 and "Internal DMA Controller Reception Sequences" on page 11–54.
   - 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" on page 11–40. 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 0xFFFFFFFA.
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.

Alternative Boot Operation for MMC4.3 Cards

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

For these cards, the software driver must perform the following steps:

1. Discover the card as described in "Enumerated Card Stack" on page 11–37.
2. Read the MMC card device's EXT_CSD registers and examine the following fields:
   - BOOT_PARTITION_ENABLE
   - BOOT_SIZE_MULT
   - BOOT_INFO

   For more information, refer to "Access to Boot Partition" in *JEDEC Standard No. JESD84-A43*, as referenced in "References" on page 11–79.
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" on page 11–74.

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.

SD/MMC Controller Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the following link:

- **sdmmc**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

References

The following industry specifications provide detailed information about the standards and protocols implemented by the SD/MMC controller.

The following specifications are available on the JEDEC website (www.jedec.org):

- *JEDEC Standard No. 84-A441—Embedded MultiMediaCard (e•MMC) e•MMC/Card Product Standard, High Capacity, including Reliable Write, Boot, Sleep Modes, Dual Data Rate, Multiple Partitions Supports, Security Enhancement, Background Operation and High Priority Interrupt (MMCA, 4.41)*
- *JEDEC Standard No. 84-A44—Embedded MultiMediaCard (e•MMC) e•MMC/Card Product Standard, High Capacity, including Reliable Write, Boot, Sleep Modes, Dual Data Rate, Multiple Partitions Supports and Security Enhancement (MMCA, 4.4)*
- *JEDEC Standard No. JESD84-A43—Embedded MultiMediaCard (eMMC) eMMC/Card Product Standard, High Capacity, including Reliable Write, Boot, and Sleep Modes (MMCA, 4.3)*

The following specifications are available on the SD Association website (www.sdcard.org):

- *Physical Layer Simplified Specification, Version 3.01—SD Specifications Part 1 Physical Layer Simplified Specification Version 3.01*
- *SDIO Simplified Specification Version 2.00—SD Specifications Part E1 SDIO Simplified Specification Version 2.00*

Document Revision History

Table 11–29 shows the revision history for this document.

Table 11–29. Document Revision History

Date	Version	Changes
November 2012	1.1	■ Added programming model section. ■ Reorganized programming information. ■ Added information about ECCs. ■ Added pin listing. ■ Updated clocks section.
January 2012	1.0	Initial release

12. Quad SPI Flash Controller

cv_54012-1.2

The hardware 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

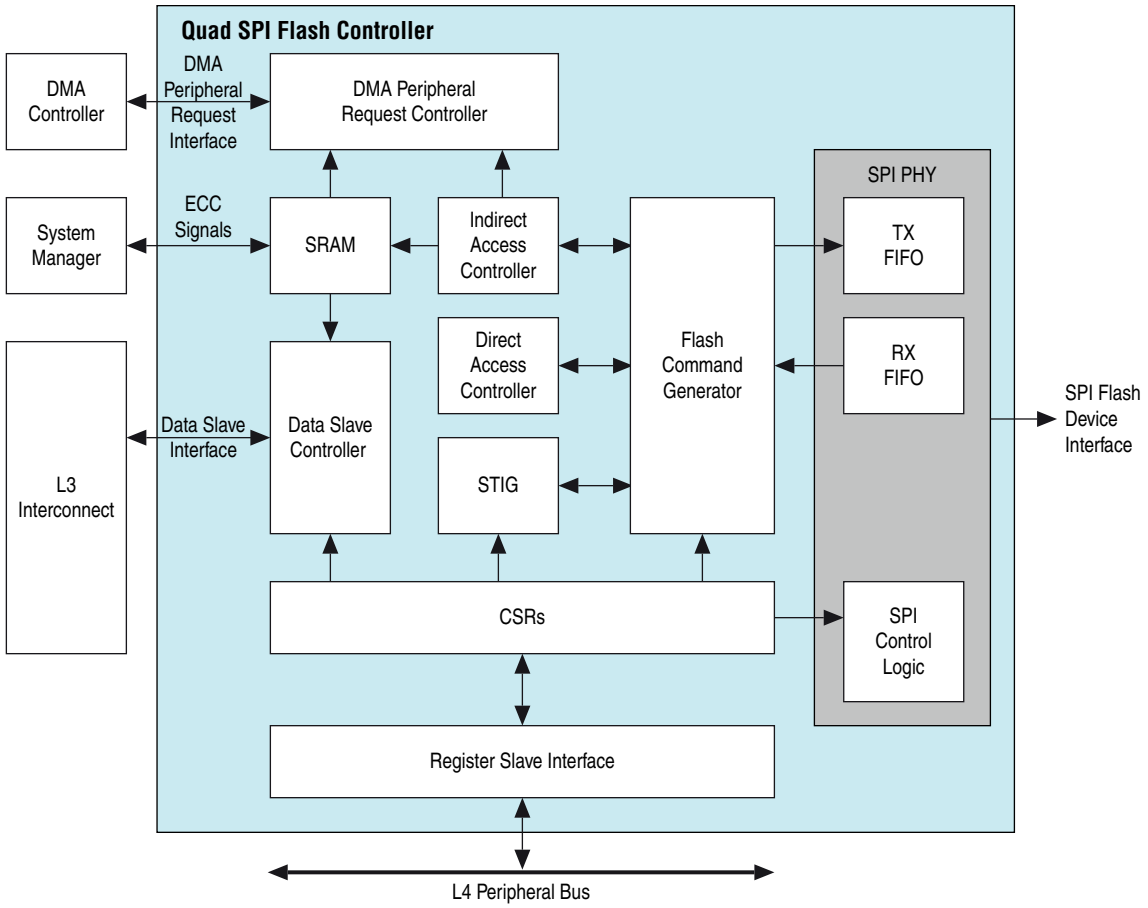
The quad SPI flash controller supports the following features:

- Single, dual, and quad I/O commands
- Device frequencies up to 108 MHz
- Direct access and indirect access modes
- External direct memory access (DMA) controller support for indirect transfers
- Configurable polarity, phase, and delay
- Programmable write-protected regions
- Local buffering with error correction code (ECC) logic for indirect transfers
- Up to four devices
- eXecute-In-Place (XIP) flash devices

Quad SPI Flash Controller Block Diagram and System Integration

Figure 12–1 shows the quad SPI flash controller block diagram.

Figure 12–1. Quad SPI Flash Controller Block Diagram



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

Functional Description of the Quad SPI Flash Controller

This section describes the functions of the quad SPI flash controller.

Overview

Terms used in this section are defined in detail in the sections that follow.

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 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. For information about these modes, refer to the following sections.

The data slave is 32 bits wide. Byte, half-word, and word accesses are permitted.

For write accesses, only incrementing bursts are supported, and only of sizes 1, 4, 8, and 16 transfers. For read accesses, all burst types and sizes are supported.

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.

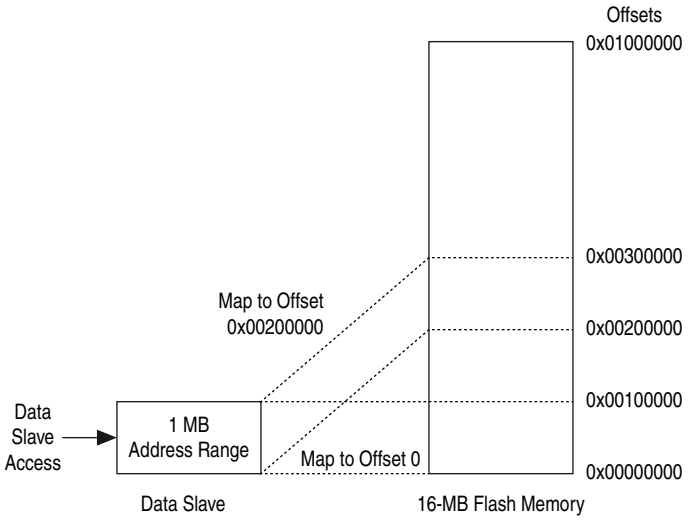
Direct Access Mode

In direct access mode, an access to the data slave triggers a read or write command to the flash memory. To use the direct access mode, enable the direct access controller with the enable direct access controller bit (`endiracc`) of the quad SPI configuration register (`cfg`).

An external master, for example a processor, triggers the direct access controller with a read or write operation to the data slave interface. The data slave exposes a 1 MB window into the flash device. You can remap this window to any 1 MB location within the flash device.

Figure 12–2 shows an example remapping.

Figure 12–2. 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 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.

The quad SPI controller does not issue any error status for accesses that lie outside the connected flash memory space.

Indirect Access Mode

In indirect access mode, flash data is temporarily buffered in the quad SPI controller's 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 (`indrd`)
- Indirect read transfer watermark register (`indrdwater`)
- Indirect read transfer start address register (`indrdstaddr`)
- Indirect read transfer number bytes register (`indrdcnt`)
- 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 `indrdstaddr` register and the total number of bytes to be fetched is specified in the `indircnt` register. Writing 1 to the start indirect read bit (`start`) of the `indrd` 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 will pad 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.

Another alternative is to use the fill level watermark of the SRAM, which you configure in the `indrdwater` register. When the SRAM fill level passes the watermark level, the indirect transfer watermark interrupt is generated. You can disable the watermark feature by writing zero to the `indrdwater` 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 `indrd` register to 1. For more information, refer to "Indirect Read Operation" on page 12–15.

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 software to write the next page of data to 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 the watermark feature by writing zero 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 "Indirect Write Operation" on page 12–16.

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.

Local Memory Buffer

The SRAM local memory buffer is a 128 by 32-bit (512 total bytes) memory and includes support for 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.

For more information, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

The SRAM has two partitions, with the lower partition reserved for indirect read operations and the upper partition for indirect write operations. The size of the partitions is specified in the SRAM partition register (`srampart`), based on 32-bit word sizes. 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.

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.

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 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 *Cyclone V Device Handbook*.

STIG Operation

The STIG provides software a method to access the flash device registers directly. 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.

Altera recommends using the STIG instead of the SPI legacy mode to access the flash device registers and perform erase operations.

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`).

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 12–1 shows how software should configure the quad SPI controller for each specific read and write instruction supported by the Micron N25Q128 device.

Table 12–1. Quad SPI Configuration for Micron N25Q128 Device

Instruction	Lanes Used By Opcode	Lanes Used to Send Address	Lanes Used to Send Data	instwidth Value	addrwidth Value	datawidth Value
Read Instructions						
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
Write Instructions						
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
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

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 "XIP Mode Operations" on page 12–17.

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 (`uppwrprot`) 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 `uppwrprt` 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. Altera 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_clk`) and one clock output (`sclk_out`). The quad SPI flash controller uses the `l4_mp_clk` clock to clock the data slave transfers and register slave accesses. The `qspi_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 `sclk_out` clock is the clock source for the connected flash devices.

The `qspi_clk` clock must be greater than two times the `l4_mp_clk`. The `sclk_out` clock is derived by dividing down the `qspi_clk` clock by the baud rate divisor field (`bauddiv`) of the `cfg` register.

For more information, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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.

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Interrupts

All interrupt sources are combined to create a single level-sensitive, active-high interrupt (`qspi_intr_in`). 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 the interrupt status register. The interrupts are individually maskable through the interrupt mask register (`irqmask`). Table 12–2 lists the interrupt sources in the `irqstat` register.

Table 12–2. 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 to zero only when the register is written to.
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.
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.

Table 12–2. Interrupt Sources in the irqstat Register

Interrupt Source	Description
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

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. Table 12–3 lists the I/O pin use of the quad SPI controller interface signals for each operation mode.

Table 12–3. 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			Serial clock

Quad SPI Flash Controller Programming Model

This section describes the programming model for the quad SPI controller.

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_clk clock.

   For example, an 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_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 uppwrprot), when write protection is required.
9. Enable required interrupts though 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) as needed. 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.

Indirect Read Operation

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 "Setting Up the Quad SPI Flash Controller" on page 12–14.
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 indrd 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 `indrd` register.

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 "Setting Up the Quad SPI Flash Controller" on page 12–14.
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 the number of bytes for single and burst type DMA transfers in the `dmaper` register.
6. Optionally set the SRAM watermark level in the `indrdwater` register to control the rate DMA requests are issued.
7. Start an indirect read access by setting the `start` field of the `indrd` 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 `indrd` register.

Indirect Write Operation

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 "Setting Up the Quad SPI Flash Controller" on page 12–14.
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. 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 "Indirect Write Operation" on page 12–6.
7. Start the indirect write operation by setting the `start` field of the `indwr` 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 space in the SRAM.
9. 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.

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 "Setting Up the Quad SPI Flash Controller" on page 12–14.
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 "Indirect Write Operation" on page 12–6.
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.

XIP Mode Operations

This section describes entering and exiting XIP mode. 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

The following sections describe the software steps to enter XIP mode for various types of flash devices.

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 will be 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 known the device will 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 known in advance whether or not the device will 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 "Entering XIP Mode" on page 12–17. Software must be aware of the mode bit requirements of the device, because XIP mode entry and exit varies by device.

Quad SPI Flash Controller Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the following links for the module instance:

- **qspiregs**
- **qspidata**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 12–4 shows the revision history for this document.

Table 12–4. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	Added block diagram and system integration, functional description, programming model, and address map and register definitions sections.
January 2012	1.0	Initial release.

Section VI. Peripherals

This section includes the following chapters:

- Chapter 13, FPGA Manager
- Chapter 14, System Manager
- Chapter 15, Scan Manager
- Chapter 16, DMA Controller
- Chapter 17, Ethernet Media Access Controller
- Chapter 18, USB 2.0 OTG Controller
- Chapter 19, SPI Controller
- Chapter 20, I²C Controller
- Chapter 21, UART Controller
- Chapter 22, General-Purpose I/O Interface
- Chapter 23, Timer
- Chapter 24, Watchdog Timer
- Chapter 25, CAN Controller

For information about the revision history for chapters in this section, refer to "Document Revision History" in each individual chapter.

13. FPGA Manager

cv_54013-1.3

The FPGA manager in the hard processor system (HPS) manages and monitors the FPGA portion of the system on a chip (SoC) FPGA 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

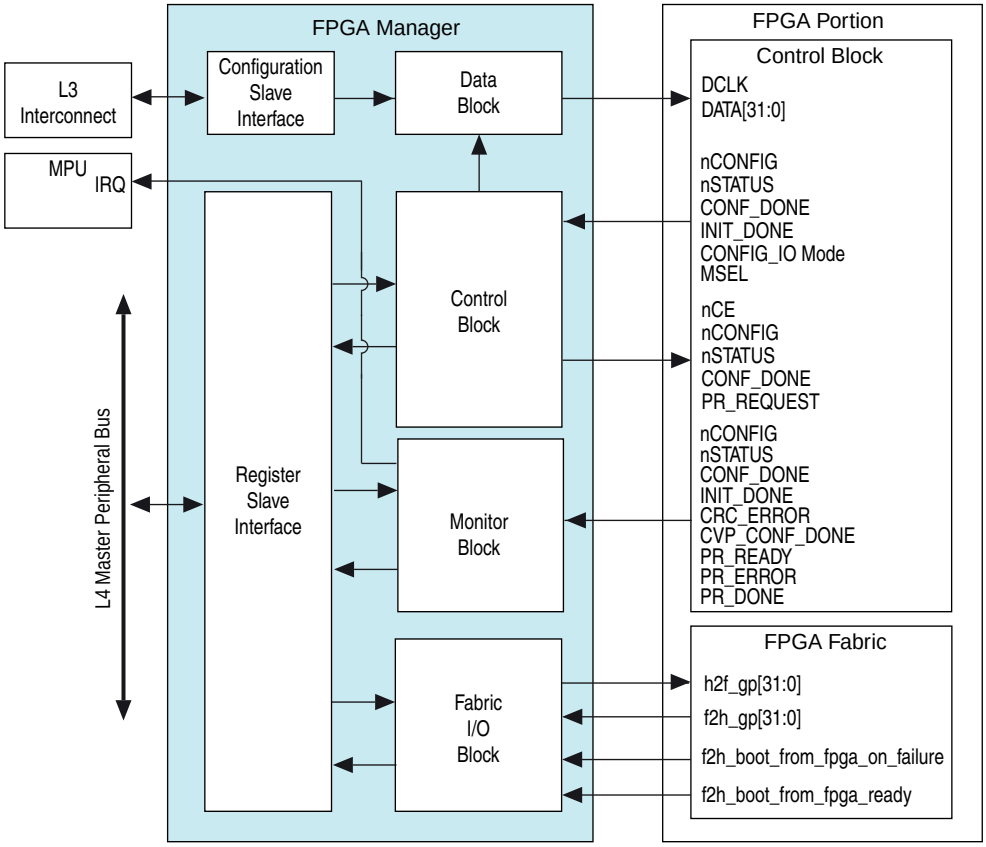
The FPGA manager provides the following functionality and features:

- 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

FPGA Manager Block Diagram and System Integration

Figure 13–1 shows a block diagram of the FPGA manager.

Figure 13–1. 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 handshaking 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 following blocks to monitor the signals coming from the FPGA portion of the device.

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 13–7.

The boot handshaking input signals from the FPGA fabric are read by reading the `misci` register. The `f2h_boot_from_fpga_ready` signal indicates to the boot ROM when logic in the FPGA fabric is ready to accept configuration interface requests from the HPS-to-FPGA bridge when the boot ROM is booting from the FPGA. 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.

Monitor

The monitor block is an instance of the Synopsys® DesignWare® GPIO 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 13–7.

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 only covers configuring the FPGA through the HPS.

For information about configuring the FPGA using an external device, refer to the *Configuration, Design Security, and Remote System Upgrades in Cyclone V Devices* chapter in volume 1 of the *Cyclone V Device Handbook*.

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. Table 13–1 lists supported `MSEL` values when the FPGA is configured by the HPS.

HPS software sets the clock-to-data ratio field (`cdratio`) and configuration data width bit (`cfgwdth`) in the control register (`ctrl`) to match the `MSEL` pins. The `cdratio` field and `cfgwdth` 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 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

The FPGA manager supports a data width of 32 or 16 bits. When configuring the FPGA fabric from the HPS, Altera recommends that you always set the data width to 32 bits. For partial reconfiguration, the 16-bit data width is the only option.

Table 13–1 lists the supported configuration schemes and their respective MSEL and control register settings when the HPS configures the FPGA.

Table 13–1. Configuration Schemes for FPGA Configuration by the HPS

Configuration Scheme	Compression Feature	Design Security Feature	POR Delay (2)	MSEL[4..0] (3)	cfgwdth	cdratio	Supports Partial Reconfiguration
FPP ×16	Disabled	AES Disabled	Fast	00000	0	1	Yes
			Standard	00100	0	1	No
	Disabled	AES Enabled	Fast	00001	0	2	Yes
			Standard	00101	0	2	No
	Enabled	Optional (1)	Fast	00010	0	4	Yes
			Standard	00110	0	4	No
FPP ×32	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 (1)	Fast	01010	1	8	No
			Standard	01110	1	8	No

Notes to Table 13–1:

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

(2) For information about POR delay, refer to the *Configuration, Design Security, and Remote System Upgrades in Cyclone V Devices* chapter in volume 1 of the Cyclone V Device Handbook.

(3) 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 Cyclone V Devices* chapter in volume 1 of the Cyclone V Device Handbook.

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

Power Up Phase

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

You must set the cdratio and cfgwdth 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.

For more information about configuring the FPGA through the HPS, refer to the *Booting and Configuration* appendix in volume 3 of the *Cyclone V Device Handbook*.

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.

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 (dclkcnt) register to drive DCLK pulses to the FPGA. Writing to the cnt field of the dclkcnt 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 (dclkstat) register is set to 1. Software polls the dcntdone bit to know when all of the DCLK pulses have been sent.

Before another write to the dclkcnt 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.

For more information about configuring the FPGA through the HPS, refer to the *Booting and Configuration* appendix in volume 3 of the *Cyclone V Device Handbook*.

For more information about configuring the FPGA in general, refer to the*Configuration, Design Security, and Remote System Upgrades in Cyclone V Devices* chapter in volume 1 of the *Cyclone V Device Handbook*.

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.

For more information about the clock manager, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Reset

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

For more information about the reset manager, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

FPGA Manager Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the following links for the module instance:

- **fpgamgrregs**
- **fpgamgrdata**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 13–2 shows the revision history for this document.

Table 13–2. Document Revision History

Date	Version	Changes
November 2012	1.3	Minor updates.
June 2012	1.2	Updated the FPGA configuration section.
May 2012	1.1	■ Updated the configuration schemes table. ■ Updated the FPGA configuration section. ■ Added address map and register definitions section.
January 2012	1.0	Initial release.

14. System Manager

cv_54014-1.2

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:

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

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.
- Freezes the I/O pins after the HPS comes out of cold reset and during serial configuration.
- Selects the EMAC level 3 (L3) master signal options.
- 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.
- Selects whether the CAN controller or the FPGA fabric can issue requests to four of the DMA controller peripheral request interfaces.
- 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.

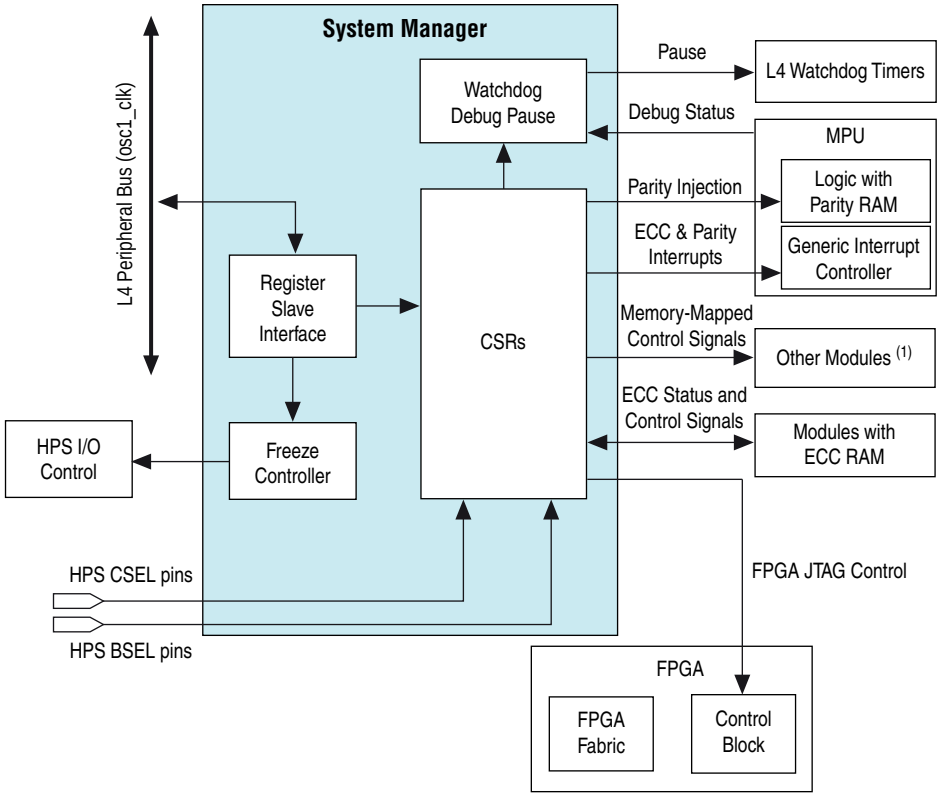
- Sends error correction code (ECC) enable signals to all HPS modules with ECC-protected RAM.
- Provides the capability to inject errors during testing in modules with ECC-protected RAM.
- Controls I/O pin multiplexing in the HPS.
- Provides information about device-specific HPS options such as the number of processor cores and the presence of CAN controllers.
- Triggers parity failures during testing in parity-protected RAMs in the MPU subsystem.
- Allows the HPS to access the FPGA JTAG controller through the scan manager for testing purposes.

System Manager Block Diagram and System Integration

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

Figure 14–1 shows a block diagram of the system manager.

Figure 14–1. System Manager Block Diagram



Note to Figure 14–1:

(1) Refer to the list of modules on the following page.

The system manager consists of the following blocks:

- CSRs—provide the following functionality:
  - Provide memory-mapped access to control signals for the following HPS modules:
    - EMACs
    - Debug core
    - Reset manager
    - SD/MMC controller
    - NAND controller
    - SPI masters
    - Quad SPI controller
    - USB controllers
    - DMA controller
    - On-chip RAM
    - CAN controllers
  - Route ECC and parity interrupts to the MPU
  - Store status information received from other HPS modules
  - Controls multiplexing between the FPGA JTAG pins and the scan manager
- Register slave interface—provides connected masters access to the CSRs in the system manager.
- Watchdog debug pause—accepts the debug mode status from the MPU subsystem and pauses the L4 watchdog timers.
- Freeze controller—responsible for placing the HPS I/O pins into a safe state so that they can be configured by software.

Functional Description of the System Manager

This section describes the functional operation 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
- Enables and controls ECC and parity in HPS modules
- Provides freeze signals to the HPS-configurable I/O pins during configuration
- Enables and disables HPS interfaces
- Controls I/O pin multiplexing
- 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. The following information is available to software:

- Sampled value of the HPS boot select (BSEL) pins
- Sampled value of the HPS clock select (CSEL) pins

The boot configuration information comes from the HPS BSEL and CSEL pins. The BSEL signal informs software which flash device CPU0 booted from. The CSEL signal informs software which clock frequency is used for the flash controllers (NAND, SD/MMC, quad SPI).

For boot and clock source values, refer to the *Booting and Configuration* appendix in volume 3 of the *Cyclone V Device Handbook*.

The system manager also provides information about the type of HPS variant available in the system on a chip (SoC) FPGA through the HPS information register (hpsinfo) so that software can determine how many processor cores are present and whether the CAN controllers are present in the HPS.

Additional Module Control

Each module in the HPS has its own CSRs, providing access to the internal state of the module. Other modules, as well as logic in the FPGA fabric, can use these CSRs to control and monitor functions in the HPS modules. The system manager CSRs provide access to additional module state information, enabling additional control and monitoring. Therefore, to fully control each module, you must manipulate both the module's own CSRs, and CSRs in the system manager. This section describes system manager CSR usage for each module.

Scan Manager

Either the FPGA JTAG pins or the scan manager can drive JTAG signals to the FPGA. Registers in the system manager control a multiplexer that determines which source drives the JTAG signals to the FPGA. Set the FPGA JTAG enable bit (fpgajtagen) of the control register (ctrl) in the scan manager group (scanmgrgrp) to control this selection.

Software running on the HPS must ensure that the FPGA JTAG pins and scan manager's connection to the FPGA control block are inactive before changing the source that drives JTAG signals to the FPGA.

For more information about the scan manager, refer to the *Scan Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

DMA Controller

There are eight DMA peripheral request interfaces. The FPGA fabric always has access to interfaces 0 to 3. Registers in the system manager control a group of multiplexers that determine whether the FPGA fabric or a CAN controller has access to each of interfaces 4 to 7. Table 14–1 lists how the channel select bits (`chansel[3:0]`) of the control register (`ctrl`) in the DMA controller group (`dmagrp`) control these selections.

Table 14–1. DMA Peripheral Request Interfaces 4 to 7 Usage

Channel Select Bit	Register Value	Usage
0	0	DMA channel 4 connects to the FPGA fabric
	1	DMA channel 4 connects to the CAN0 controller
1	0	DMA channel 5 connects to the FPGA fabric
	1	DMA channel 5 connects to the CAN0 controller
2	0	DMA channel 6 connects to the FPGA fabric
	1	DMA channel 6 connects to the CAN1 controller
3	0	DMA channel 7 connects to the FPGA fabric
	1	DMA channel 7 connects to the CAN1 controller

The security state of the DMA controller is controlled by the manager thread security (`mgrnonsecure`) and interrupt security (`irqnonsecure`) bits of the `ctrl` register, all in the `dmagrp` group.

The security state of each DMA peripheral request interface is controlled by the peripheral nonsecure bits (`nonsecure[31:0]`) of the peripheral security register (`persecurity`) in the `dmagrp` group. The DMA controller samples these register bits when it is brought out of reset.

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

For more information about the DMA controller, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

NAND Flash Controller

Software uses the bootstrap control register (`bootstrap`) in the NAND flash controller group (`nandgrp`) to modify the default behavior of the NAND flash controller after reset. The NAND flash controller samples the register bits when it is brought out of reset.

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

- Bootstrap inhibit initialization bit (`noinit`)—inhibits the NAND flash controller from performing initialization 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 of 512 byte page size is connected to the system.

- Bootstrap inhibit load block 0 page 0 bit (noloadb0p0)—inhibits the NAND flash controller from loading page 0 of block 0 of the NAND flash device as part of 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.

Registers in the system manager control the L3 master ARCACHE and AWCACHE signals. Set the NAND arcache (arcache[0]) and NAND awcache (awcache[0]) bits of the NAND L3 master AxCACHE register (l3master) in the nandgrp group to control these selections. These bits define the cache attributes for the master transactions of the DMA engine in the NAND controller.

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

For more information about the NAND flash controller, refer to the *NAND Flash Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

EMAC

The system manager allows software to select either osc1_clk or fpga_ptp_ref_clk as the source of the IEEE 1588 reference clock for each EMAC. Set the PTP clock select (ptpclksel[0]) and (ptpclksel[1) bits of the ctrl register in the EMAC group (emacgrp) to control this selection.

For more information about the reference clock, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Registers in the system manager control the L3 master ARCACHE and AWCACHE signals. Set the arcache[0], arcache[1], awcache[0], and awcache[1]bits of the EMAC L3 master AxCACHE register (l3master) in the emacgrp group to control these selections. These bits define the cache attributes for the master transactions of the DMA engine in the EMAC controllers.

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

For more information about the EMAC, refer to the *Ethernet Media Access Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

USB 2.0 OTG Controller

Registers in the system manager control whether the USB 2.0 OTG controller L3 master is used for data or opcode access. Set the USB HPROT data/opcode bits (hprotdata[0] and hprotdata[1]) of the USB L3 master HPROT register (l3master) in the USB controller group (usbgrp) to specify the access for each USB controller.

You can also use the system manager to specify whether the USB controller L3 master access is privileged, bufferable, or cacheable for each USB controller by setting the USB HPROT privileged (hprotpriv[0] and hprotdata[1]), USB HPROT bufferable (hprotbuff[0] and hprotbuff[1]), and USB HPROT cacheable (hprotcache[0] and hprotcache[1]) bits of the l3master register in the usbgrp group.

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

For more information about the USB 2.0 OTG controller, refer to the *USB 2.0 OTG Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

SD/MMC Controller

Registers in the system manager control whether the SD/MMC controller L3 master is used for data or opcode access. Set the SD/MMC `hprotdata[0]` bit of the SD/MMC `l3master` register in the SD/MMC controller group (`sdmmcgrp`) to specify the access mode.

You can use the system manager to specify whether the SD/MMC controller L3 master access is privileged, bufferable, or cacheable by setting the SD/MMC `hprotpriv[0]`, `hprotbuff[0]`, and `hprotcache[0]` bits of the `l3master` register in the `sdmmcgrp` group.

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

The system manager allows 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 (`smplsel`) bits of the `ctrl` register in the `sdmmcgrp` group. You can also select which feedback clock (`fb_clk`) to use as the `cclk_in_sample` clock. Set the feedback clock select (`fbclksel`) bit of the `ctrl` register to control this selection.

For more information about the SD/MMC controller, refer to the *SD/MMC Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 (`mode[0]` and `mode[1]`) bits of the L4 watchdog debug register (`wddbg`). Each watchdog timer built into the MPU subsystem is automatically paused when its associated CPU enters debug mode.

For more information about the watchdog timer, refer to the *Watchdog Timer* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 (`warmrstcfgpinmux`) of the `ctrl` register in the boot ROM code register group (`romcodegrp`) to enable or disable this control.

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 IOs for boot pins bit (warmrstcfgio) of the ctrl register in the romcodegrp group to enable or disable this control. 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 jumps to that address to pass control to software.

The preloader state register (initswstate) stores the magic number 0x49535756 written by the preloader to indicate there is a valid preloader software image in the on-chip RAM.

There can be up to four preloader images stored in flash memory. The preloader last image loaded register (initswlastld) contains the index of the preloader image 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 registers in the warm boot from on-chip RAM group (warmramgrp) in the romcodegrp group are used by the boot ROM code to support booting from the on-chip RAM on a warm reset. Table 14–2 lists the registers and their purposes.

Table 14–2. The warmramgrp Registers

Register	Name	Purpose
enable	Enable	Controls whether the boot ROM attempts to boot from the contents of the on-chip RAM on a warm reset.
datastart	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.
length	Length	Contains the length in bytes of the region in the on-chip RAM available for warm boot CRC validation.
execution	Execution offset	Contains the byte offset into the on-chip RAM that the boot code jumps to if the CRC validation succeeds.
crc	Expected CRC	Contains the expected CRC of the region in the on-chip RAM.

All the registers in Table 14–2 must be written by software prior to the warm reset occurring.

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 in the romhwgrp group. 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 of the ctrl register in the romhwgrp group to add an additional wait state to the read access of the boot ROM. The enable safe mode warm reset update bit (ensfmdwru) of the ctrl register controls whether the waitstate bit is updated during a warm reset.

Freeze Controller

The freeze controller provides freeze signals to the HPS configurable I/O elements. These freeze signals ensure that the I/O pins are in a safe state upon FPGA power on and until they are configured.

The freeze controller divides the HPS configurable I/O pins into four groups. Each group has an independent set of freeze signals, referred to as a freeze channel, which is driven by the freeze controller. Separate freeze channels allow each group of I/O pins to be independently frozen, configured, and thawed.

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 14–3 lists the I/O banks and bank types that each freeze channel controls.

Table 14–3. Freeze Channels and HPS I/O Banks

Freeze Channel	Bank Type	HPS I/O Bank
0	VIO	I/O bank 7D and bank 7E
1	VIO	I/O bank 7B and bank 7C
2	VIO	I/O bank 7A
3	HIO	I/O bank 6

When the I/O elements are frozen, the configuration registers in the I/O elements are not placed in a reset state. Instead, the outputs of the configuration registers are gated off by the freeze signals until software running on the HPS thaws the I/O elements. The scan manager must configure the I/O elements in each group before software thaws them.

While frozen, the HPS I/O pins are in the following state:

- I/O buffers are in tri-state mode
- Weak pull-up is enabled
- Outputs of configuration registers are gated
- Internal registers are reset to their initial states
- Bus hold is in tri-state mode
- On-chip termination (OCT) internal timer is reset

Software can generate HPS I/O freeze signals by setting the VIO control (`vioctrl`) and HIO control (`hioctrl`) registers in the freeze control group (`frzctrl`). The system manager allows the following options to be set in the HPS VIO and HIO banks:

- Enable or disable I/O pin configuration
- Allow I/O pin configuration to control the bus hold circuit
- Allow I/O pin configuration to control I/O tri-state
- Allow I/O pin configuration to control weak pull-up resistor
- Allow I/O pin configuration to control the slew-rate
- Reset registers in the delay-locked loop (DLL)
- Reset registers in the OCT

- Reset registers in the I/O elements and data strobe (DQS)
- Disable I/O pin configuration in the OCT calibration block, or start OCT calibration state machine and enable I/O pin configuration in the OCT calibration block

Software controls each freeze channel through a set of memory-mapped control register fields in the system manager. The freeze signals for VIO channel 1 is controlled either by a software state machine, or by a hardware state machine in the freeze controller. Software chooses between the two freeze signal sources by setting the source register (`src`) in the `frzctrl` group.

The system manager initiates a freeze or thaw operation on VIO channel 1 through the hardware state machine. Freeze or thaw is initiated by requesting the hardware state machine to generate a freeze signal sequence. This sequence transitions between frozen and thawed states by setting the VIO channel 1 freeze/thaw request bit (`vio1req`) of the hardware control register (`hwctrl`) in the `frzctrl` group.

The system manager allows software to determine the current state of the VIO channel 1 (frozen or thawed) or to determine when a freeze or thaw request is made by reading the VIO channel 1 state bit (`vio1state`) of the `hwctrl` register in the `frzctrl` group.

FPGA Interface Enables

The system manager can enable or disable interfaces between the FPGA and HPS. The interfaces must be disabled when not in use to avoid undefined behavior.

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

☞ Ensure that all interfaces between the FPGA and HPS are inactive before disabling them.

You can set the individual disable register (`indiv`) in the `fpgaintfgrp` group to disable the following interfaces between the FPGA and HPS:

- Reset request interface
- JTAG enable interface
- I/O configuration interface
- Boundary scan interface
- Trace interface
- System Trace Macrocell (STM) interface
- Cross-trigger interface (CTI)

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
- CAN controller RAM
- NAND flash controller RAM
- Quad SPI flash controller RAM
- SD/MMC controller RAM

With the exception of L2 cache data RAM, each of the ECC memories generates a single- and double-bit interrupt to the global interrupt controller (GIC) through the system manager. Set the registers in the ECC management register group (`eccgrp`) to enable or disable the ECC. The L2 cache generates the L2 cache data RAM interrupt directly.

The system manager can inject single-bit or double-bit errors into each of the ECC memories for testing purposes. Set the bits in the appropriate memory enable register to inject errors. For example, to inject a single bit EEC error into the USB0 module, set the `injs` bit of the USB0 RAM ECC enable register (`usb0`) in the ECC management register group (`eccgrp`).

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

Pin Multiplexing Control

In the HPS, many of the pins are available for use by as many as four peripherals. The system manager allows software to control pin multiplexing selections to select which peripheral in the HPS has access to each shared pin. Some pins also provide boot source and clock source information, which are sampled upon deassertion of a cold reset event. Set the registers in the pin multiplexing control group (`pinmuxgrp`) to control this selection.

☞ Do not modify the pin multiplexing selection registers after I/O configuration.

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 (`handoff`) array in the preloader register group (`iswgrp`). These registers are reset only by a cold reset.

Clocks

The system manager is driven by the `osc1_clk` clock generated by the clock manager.

For more information about the clock manager, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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

Table 14–4 lists the registers and groups of registers that are reset by a cold or warm reset.

Table 14–4. Registers and Corresponding Reset SIgnal

Register or Group	Cold Reset	Warm Reset
fpgaintfgrp	✓	—
scanmgrgrp	✓	—
frzctrl	✓	—
emacgrp	✓	✓
dmagrp	✓	✓
sdmmcgrp	✓	✓
nandgrp	✓	✓
usbgrp	✓	✓
eccgrp	✓	—
pinmuxgrp	✓	—
wddbg	✓	—
parityinj	✓	✓
romcodegrp	✓	—
romhwgrp	✓	—

For more information about the reset manager, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

System Manager Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for the following module instance:

- **sysmgr**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 14–5 shows the revision history for this document.

Table 14–5. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	Added functional description, address map and register definitions sections.
January 2012	1.0	Initial release.

15. Scan Manager

cv_54015-1.2

The scan manager is used to configure and manage the hard processor system (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.

For more information about the ARM JTAG-AP, refer to the *ARM Debug Interface v5 Architecture Specification*, which you can download from the ARM website (infocenter.arm.com).

Features of the Scan Manager

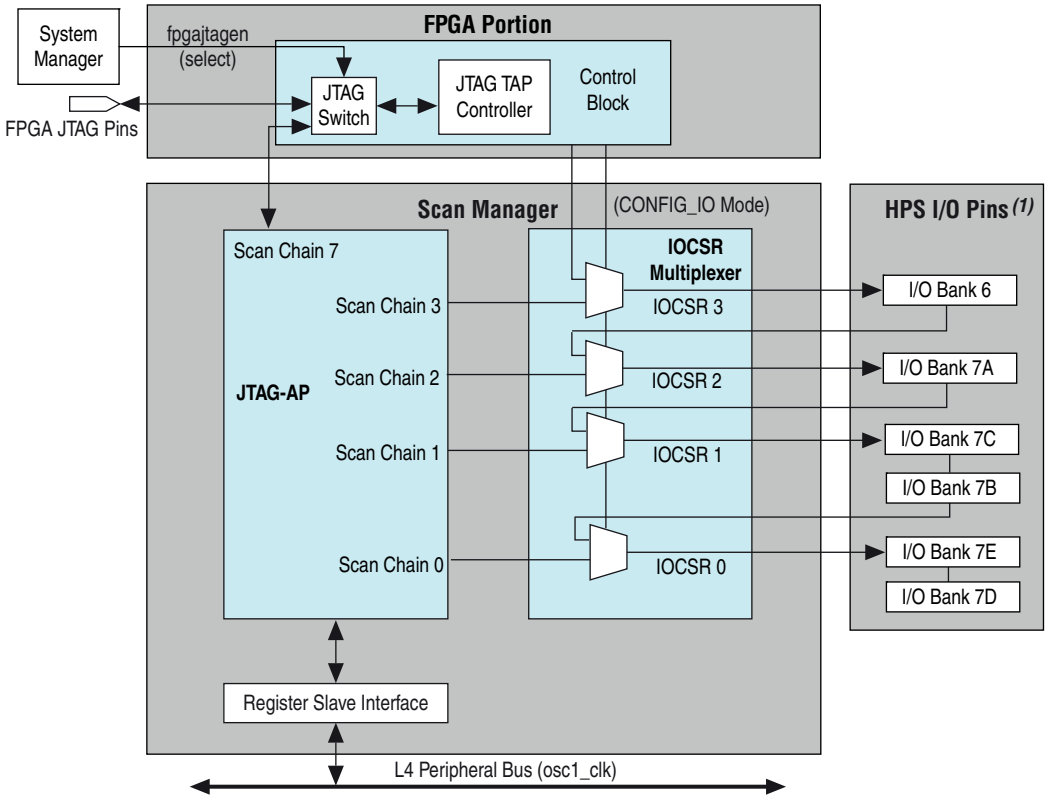
The scan manager has the following features:

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

Scan Manager Block Diagram and System Integration

Figure 15–1 shows a block diagram of the scan manager, showing how it is integrated in the SoC device.

Figure 15–1. Scan Manager Block Diagram



Note to Figure 15–1:

(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.

Table 15–1 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 Figure 15–1. The signal, register, and field names listed in the table match the names used in the *ARM Debug Interface v5 Architecture Specification*. The remainder of this chapter uses register and field names as listed in the Altera SoC register documentation. Refer to Table 15–3 to cross-reference the two sets of register names.

Table 15–1. ARM JTAG-AP Signal Use in the Scan Manager (Part 1 of 2)

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.

Table 15–1. ARM JTAG-AP Signal Use in the Scan Manager (Part 2 of 2)

Signal	Direction	Implementation
PORTENABLED[7:0]	Input	Tied to 0x8F, so all connected ports are always considered powered on. The PSTA register does not contain a useful value, so there is no reason for software to access it. Software does not need to monitor the status of ports 0-3 because 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" on page 15–5.

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 "Communicating with the JTAG TAP Controller" on page 15–5.

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 "Configuring HPS I/O Scan Chains" on page 15–4.

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 (TDI JTAG signal) and serial clock (TCK 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 15–2 shows the mapping of the IOCSR scan chains to the I/O banks.

Table 15–2. Bank Usage of IOCSR Scan Chains

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, I^2C, and CAN
3	HIO	I/O bank 6	SDRAM DDR

When the FPGA JTAG TAP controller is in CONFIG_IO mode, the controller can override the scan manager JTAG-AP and configure the HPS I/O pins. For more information, refer to "Configuring HPS I/O Scan Chains" on page 15–4.

CONFIG_IO mode is commonly used to configure the I/O pin properties prior to performing boundary scan testing.

Functional Description of the Scan Manager

This section describes the functional operation of the scan manager. The scan manager serves the following two 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.

The HPS I/O pins need to be frozen before configuring them. For more information, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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® II 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 IO bank, the corresponding scan chain must be enabled by writing to the bits in en register. The scan manager must not be active during this process. Software reads the active bit of stat register to determine the scan manager state.

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

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 `fpgajtag` field of the `en` register in the scan manager. For more information, refer to "Scan Manager Address Map and Register Definitions" on page 15–6.
- 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 `fpgajtagen` 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.

  For more information, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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.

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.

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.

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.

For details about the byte command protocol, refer to the JTAG-AP chapter in the *ARM Debug Interface v5 Architecture Specification*, which you can download from the ARM website (infocenter.arm.com).

Clocks

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

For more information, including minimum and maximum clock frequencies, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 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 spi_m_clk / 2 and runs at a maximum frequency of 100 MHz.

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.

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.

For more information about reset handshaking, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Scan Manager Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for the following module instance:

- **scanmgr**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

To improve clarity regarding how Altera uses the JTAG-AP, the ARM register names are changed in the SoC device. Table 15–3 cross references the ARM and Altera names.

Table 15–3. JTAG-AP Register Names

Altera Name	ARM Name
stat	CSW (control/status word)
en	PSEL
fifosinglebyte	BWFIFO1 for writes, BRFIFO1 for reads
fifodoublebyte	BWFIFO2 for writes, BRFIFO2 for reads
fifotriplebyte	BWFIFO3 for writes, BRFIFO3 for reads
fifoquadbyte	BWFIFO4 for writes, BRFIFO4 for reads

For more information about the ARM JTAG-AP, refer to the *ARM Debug Interface v5 Architecture Specification*, which you can download from the ARM website (infocenter.arm.com).

Document Revision History

Table 15–4 shows the revision history for this document.

Table 15–4. Document Revision History

Date	Version	Changes
November 2012	1.2	Added JTAG-AP descriptions.
May 2012	1.1	Added block diagram and system integration, functional description, and address map and register definitions sections.
January 2012	1.0	Initial release.

16. DMA Controller

cv_54016-1.1

This chapter describes the direct memory access controller (DMAC) contained in the hard processor system (HPS). The DMA controller is used to transfer 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).

For more information about ARM's DMA-330 controller, refer to the *CoreLink DMA Controller DMA-330 Revision: r1p1 Technical Reference Manual*, available from the ARM website (infocenter.arm.com).

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 memory-to-memory, memory-to-peripheral, and peripheral-to-memory transfers. The DMAC supports up to eight logical channels for different levels of service requirements. It provides up to 31 peripheral handshake interfaces for peripheral hardware flow control.

The DMA controller contains an instruction processing block that enables it to process program code that controls a DMA transfer. It also contains 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. The AXI master interface is used to perform DMA data transfer as well. 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. It also contains a multi-FIFO (MFIFO) data buffer that it uses to store data that it reads, or writes, during a DMA transfer.

The DMAC provides 11 interrupt outputs to enable efficient communication of events to the MPU subsystem. 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. Since the HPS supports some peripherals that do not comply with ARM DMA peripheral interface protocol, adapters are added to allow these peripherals to work with the DMAC. The following peripheral interface protocols are supported:

- Synopsys protocol
  - Serial peripheral interface (SPI)
  - Universal asynchronous receiver/transmitter (UART)
  - Inter-integrated circuit (I^2C)
  - FPGA

- ARM protocol
  - Quad SPI flash controller
  - System trace macrocell (STM™)
- Bosch protocol
  - Controller area network (CAN)

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.

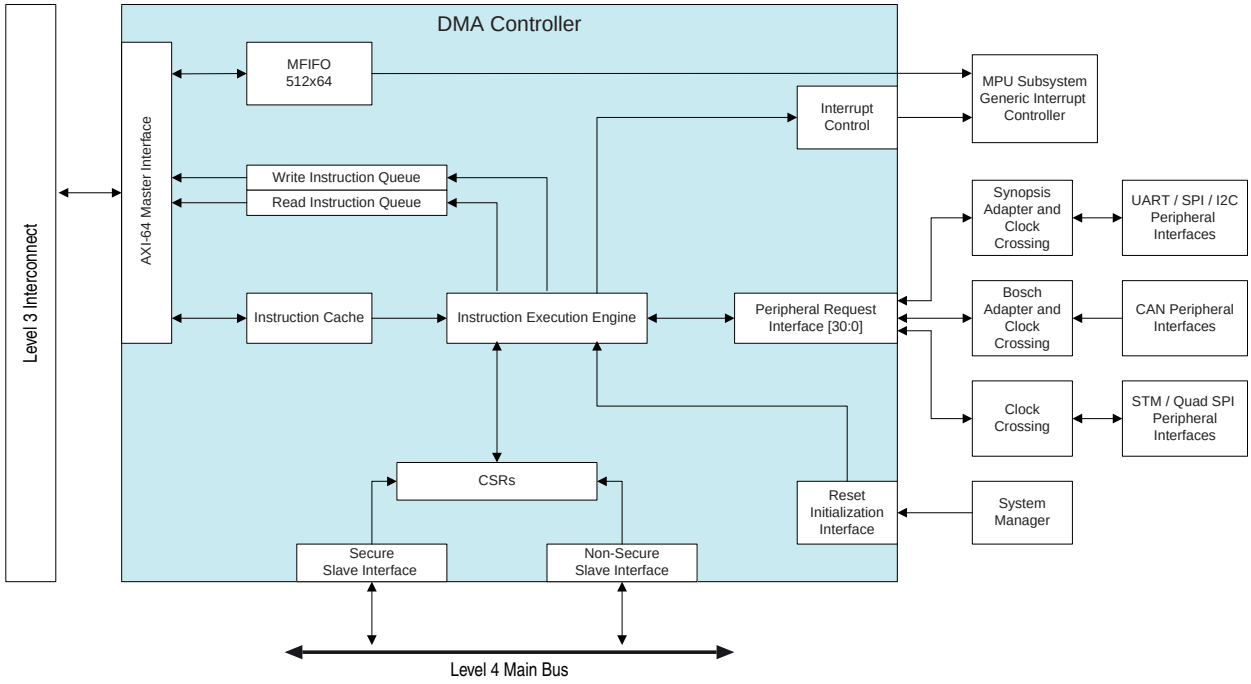
The DMAC has the following features:

- 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
  - 1 for DMA thread abort
  - 8 for events
  - 2 for MFIFO buffer ECC
- Supports 31 peripheral request interfaces
  - 4 for FPGA
  - 4 shared for FPGA or CAN
  - 8 for I^2C
  - 8 for SPI
  - 2 for quad SPI
  - 1 for System Trace Macrocell
  - 4 for UART

DMA Controller Block Diagram and System Integration

Figure 16–1 shows a block diagram of the DMAC and how it is integrated into the rest of the HPS system.

Figure 16–1. DMA Controller Connectivity



The DMA controller is driven by the `l4_main_clk` clock, which is used for the controller logic as well as 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. The DMA controller implements TrustZone® secure technology with one interface operating in the Secure state and the other operating in the Non-secure state.

Functional Description of the DMA Controller

This section describes the major interfaces and components of the DMAC, and how it operates.

Overview

The DMAC contains an instruction processing block that enables it to process program code that controls 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 is configured with eight DMA channels, with each channel capable of supporting 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 up to one instruction per clock cycle. To ensure that it regularly executes each active thread, it alternates by processing the DMA manager thread and then a DMA channel thread. It uses 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. 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 uses the AXI master interface to perform a cache line fill. If an instruction spans the end of a cache line, the DMAC performs multiple cache accesses to fetch the instruction.

When a cache line fill is in progress, the DMAC enables other threads to access the cache, but if another cache miss occurs, this stalls the pipeline 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 that it uses to store data that it reads, or writes, during a DMA transfer.

The DMAC provides multiple interrupt outputs to enable efficient communication of events to external microprocessors. 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 a microprocessor.

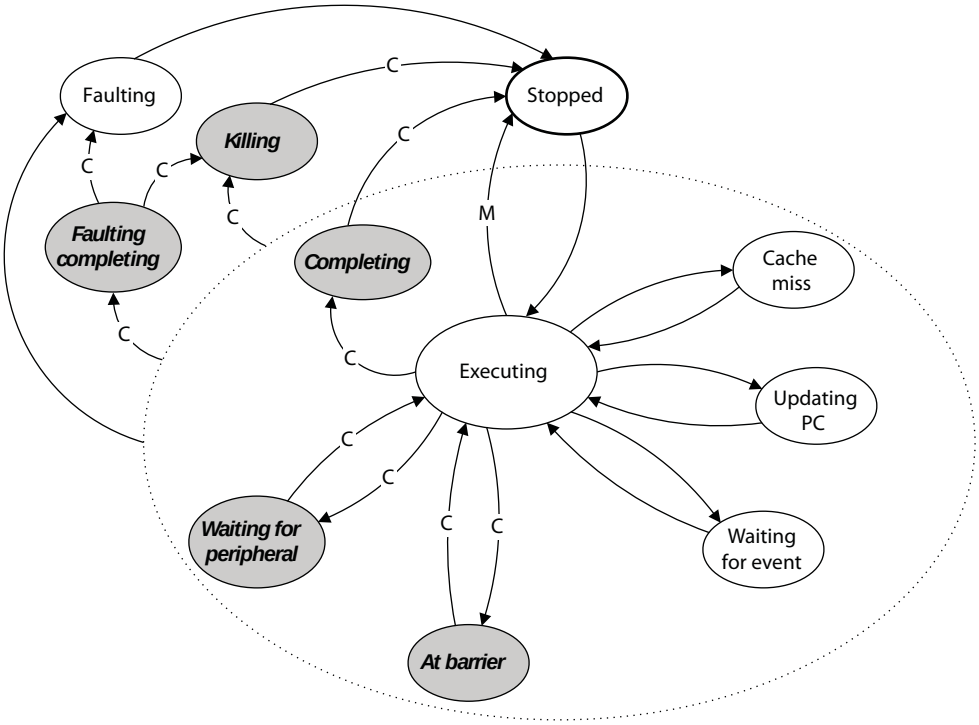
Dual slave interfaces enable the operation of the DMAC to be partitioned into the Secure state and Non-secure states. You can use the slave interfaces to access status registers and also directly execute instructions in the DMAC.

Operating States

Figure 16–2 shows the transitions among operating states for the DMA manager thread and DMA channel threads.

The DMAC provides a separate state machine for each thread.

Figure 16–2. Thread Operating States



In Figure 16–2, the DMAC permits that:

- Only DMA channel threads can use states shown in gray
- Arcs with no letter designator indicate state transitions for the DMA manager and DMA channel threads, otherwise use is restricted as follows:
  - C—DMA channel threads only.
  - M—DMA manager thread only.
- States within the dotted line can transition to the Faulting completing, Faulting, or Killing

After the DMAC exits from reset, it sets all DMA channel threads to the Stopped state, and DMA manager thread moves to the Stopped state.

The following sections describe the states:

- "Stopped" on page 16–6
- "Executing" on page 16–6
- "Cache Miss" on page 16–7
- "Updating PC" on page 16–7
- "Waiting For Event" on page 16–7
- "At Barrier" on page 16–7
- "Waiting For Peripheral" on page 16–7

- "Faulting Completing" on page 16–7
- "Faulting" on page 16–7
- "Killing" on page 16–7
- "Completing" on page 16–7

Stopped

The thread has an invalid PC and it is not fetching instructions. Depending on the thread type, you can cause the thread to move to the Executing state by:

- DMA manager thread—Issuing the DMAGO instruction through the slave interface.
- DMA channel thread—Programming the DMA manager thread to execute `DMAGO` for a DMA channel thread in the Stopped state.

Executing

The thread has a valid PC and therefore the DMAC includes the thread when it arbitrates. The thread can then change to one of the following states under the following conditions:

- Stopped—When the DMA manager thread executes `DMAEND`.
- Cache miss—When the instruction cache does not contain the next instruction for either the DMA manager thread or the DMA channel thread.
- Updating PC—When the DMAC calculates the address of the next access in the cache.
- Waiting for event—When a thread executes `DMAWFE`.
- At barrier—When a DMA channel thread either:
  - Executes `DMARMB`, `DMAWMB`, or `DMAFLUSHP`
  - Updates control registers that affect alignment, refer to "Updating DMA Channel Control Registers During a DMA Cycle" on page 16–25.
- Waiting for peripheral—When a DMA channel thread executes `DMAWFP`.
- Killing—When a DMA channel thread executes `DMAKILL`.
- Faulting completing—For a DMA channel thread when either:
  - The thread executes an undefined or invalid instruction
  - An AXI error occurs during an instruction fetch or data transfer.
- Faulting—For the DMA manager thread when either:
  - The thread executes an undefined or invalid instruction
  - An AXI error occurs during an instruction fetch.

  For a DMA channel thread when a watchdog timeout abort occurs.
- Completing—When a DMA channel thread executes `DMAEND`.

Cache Miss

The thread is stalled and the DMAC is performing a cache line fill. After it completes the cache fill, the thread returns to the Executing state.

Updating PC

The DMAC is calculating the address of the next access in the cache. After it calculates the PC, the thread returns to the Executing state.

Waiting For Event

The thread is stalled and is waiting for the DMAC to execute `DMASEV` using the corresponding event number. After the corresponding event occurs, the thread returns to the Executing state.

At Barrier

A DMA channel thread is stalled and the DMAC is waiting for transactions on the AXI to complete. After the AXI transactions complete, the thread returns to the Executing state.

Waiting For Peripheral

A DMA channel thread is stalled and the DMAC is waiting for the peripheral to provide the requested data. After the peripheral provides the data, the thread returns to the Executing state.

Faulting Completing

A DMA channel thread is waiting for the AXI master interface to signal that the outstanding load or store transactions are complete. After the transactions complete, the thread moves to the Faulting state.

Faulting

The thread is stalled indefinitely. The thread moves to the Stopped state when you use the `DBGCMD` register to instruct the DMAC to execute `DMAKILL` for that thread.

Killing

A DMA channel thread is waiting for the AXI master interface to signal that the outstanding load or store transactions are complete. After the transactions complete, the thread moves to the Stopped state.

Completing

A DMA channel thread is waiting for the AXI master interface to signal that the outstanding load or store transactions are complete. After the transactions complete, the thread moves to the Stopped state.

Initializing the DMAC

The DMAC provides several memory-mapped control signals that initialize its operating state when it exits from reset. The DMAC is configured so that it does not automatically begin executing code when it exits from reset. The three memory-mapped control signals are controlled by the system manager.

For further information, refer to the *System Manager* chapter in volume 3 of the *Cyclone® V Device Handbook*.

How to Set the Security State of the DMA Manager

The `boot_manager_ns` signal is the only method to set the security state of the DMA manager.

When the DMAC exits from reset, it reads the status of the `boot_manager_ns` signal and sets the security of the DMA manager.

When set, the security state remains constant until a state transition on the `dma_rst_n` signal resets the DMAC.

Refer to "DMA Manager Thread in Secure State" on page 16–20 and "DMA Manager Thread in Non-Secure State" on page 16–21 for a description of how the security state of the DMA manager affects how the DMAC operates.

How to Set the Security State for the Interrupt Outputs

The DMAC provides the `boot_irq_ns[7:0]` signals to enable you to assign each `irq[x]` signal to a security state.

The `boot_irq_ns[7:0]` signals are connected to the system manager. Before taking the DMA out of reset, you should program `boot_irq_ns[7:0]` through the system manager to control which interrupt bits are secure.

The DMAC samples these bits immediately after it comes out of reset, and then ignores them until the next reset.

When set, the security state of each `irq[x]` remains constant until a state transition on the `dma_rst_n` signal resets the DMAC.

Refer to "Security Usage" on page 16–20 for a description of how the security state of the `irq[x]` signals affects how the DMAC executes the `DMAWFE` and `DMASEV` instructions.

How to Set the Security State for a Peripheral Request Interface

The DMAC provides the `boot_periph_ns[31:0]` signals to enable you to assign each peripheral request interface to a security state.

The `boot_periph_ns[31:0]` signals are connected to the system manager. Before taking the DMA out of reset, you should program the `boot_periph_ns[31:0]` signals through the system manager to control which peripheral interfaces are secure.

The DMAC samples these bits immediately after it comes out of reset, and then ignores them until the next reset. When set, the security state of each peripheral request interface remains constant until a state transition on the `dma_rst_n` signal resets the DMAC.

Refer to "Security Usage" on page 16–20 for how the security state of the peripheral request interfaces affects how a DMA channel thread executes the `DMAWFP`, `DMALDP`, `DMASTP`, or `DMAFLUSHP` instructions.

Using the Slave Interfaces

The slave interfaces connect the DMAC to the level 4 (L4) main bus and enable a microprocessor to access the registers. Using these registers, a microprocessor can:

- Access the status of the DMA manager thread
- Access the status of the DMA channel threads
- Enable or clear interrupts
- Enable events
- Issue an instruction for the DMAC to execute by programming the following debug registers:
  - `DBGCMD` register
  - `DBGINST0` register
  - `DBGINST1` register

Issuing Instructions to the DMAC using a Slave Interface

When the DMAC is operating, you can only issue the following limited subset of instructions:

- `DMAGO`—Starts a DMA transaction using a DMA channel that you specify.
- `DMASEV`—Signals the occurrence of an event, or interrupt, using an event number that you specify.
- `DMAKILL`—Terminates a thread.

You must ensure that you use the appropriate slave interface, depending on the security state in which the `boot_manager_ns` signal initializes the DMAC to operate. For example, if the DMAC is in the Secure state, you must issue the instruction using the secure slave interface, otherwise the DMAC ignores the instruction. You can use the secure or non-secure slave interface to start or restart a DMA channel when the DMAC is in the Non-secure state.

Before you can issue instructions using the debug instruction registers or the `DBGCMD` register, you must read the `DBGSTATUS` register to ensure that debug is idle, otherwise the DMAC ignores the instructions.

When the DMAC receives an instruction from a slave interface, it can take several clock cycles before it can process the instruction, for example, if the pipeline is busy processing another instruction.

Prior to issuing DMAGO, you must ensure that the system memory contains a suitable program for the DMAC to execute, starting at the address that the DMAGO specifies.

Using DMAGO with the Debug Instruction Registers

This example shows the necessary steps to start a DMA channel thread using the debug instruction registers.

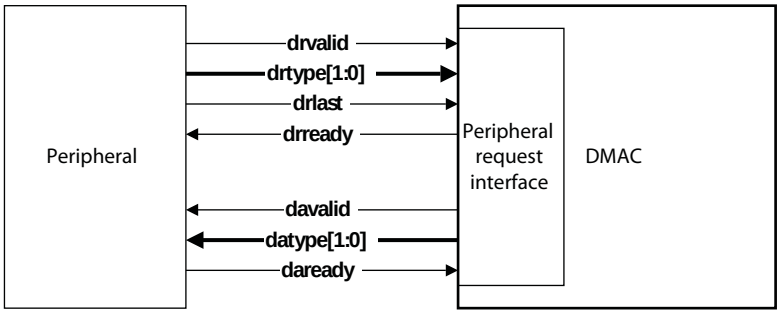
1. Create a program for the DMA channel.
2. Store the program in a region of system memory.
3. Use one of the slave interfaces on the DMAC to program a DMAGO instruction as follows:
   a. Poll the DBGSTATUS register to ensure that debug is idle, that is, the dbgstatus bit is 0.
   b. Write to the DBGINST0 register and enter the:
      - Instruction byte 0 encoding for DMAGO.
      - Instruction byte 1 encoding for DMAGO.
      - Debug thread bit to 0. This selects the DMA manager thread.
   c. Write to the DBGINST1 register with the DMAGO instruction byte [5:2] data. You must set these four bytes to the address of the first instruction in the program, that is written to system memory in step b.
4. Instruct the DMAC to execute the instruction that the debug instruction registers contain by writing zero to the DBGCMD register. The DMAC starts the DMA channel thread and sets the dbgstatus bit to 1. After the DMAC completes execution of the instruction, it clears the dbgstatus bit to 0.

Peripheral Request Interface

Figure 16–3 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 16–3. Request and Acknowledge Buses on the Peripheral Request Interface



The peripheral signals the following on the request bus:

- Request a single transfer
- Request a burst transfer
- Acknowledge a flush request.

The peripheral signals the DMAC when it issues the last request of the DMA transfer sequence.

The DMAC can signal the following on the acknowledge bus:

- Signal when it completes the requested single transfer
- Signal when it completes the requested burst transfer
- Issue a flush request.

The DMAC allows you to 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. For more information, refer to "DMAWFP" on page 16–43.

The DMAC supports 31 peripheral request handshakes. Each request handshake can receive up to four outstanding requests, and is assigned a specific peripheral device ID. Table 16–1 lists the peripheral device ID assignments.

Table 16–1. Peripheral Request Interface Mapping

Peripheral	Request Interface ID	Protocol	Peripheral	Request Interface ID	Protocol
FPGA 0	0	Synopsys	SPI Master 0 TX	16	Synopsys
FPGA 1	1	Synopsys	SPI Master 0 RX	17	Synopsys
FPGA 2	2	Synopsys	SPI Slave 0 TX	18	Synopsys
FPGA 3	3	Synopsys	SPI Slave 0 RX	19	Synopsys
FPGA 4 / CAN 0 interface 1	4	Synopsys / Bosch	SPI Master 1 TX	20	Synopsys
FPGA 5 / CAN 0 interface 2	5	Synopsys / Bosch	SPI Master 1 RX	21	Synopsys
FPGA 6 / CAN 1 interface 1	6	Synopsys / Bosch	SPI Slave 1 TX	22	Synopsys
FPGA 7 / CAN 1 interface 2	7	Synopsys / Bosch	SPI Slave 1 RX	23	Synopsys
I^2C 0 TX	8	Synopsys	Quad SPI Flash TX	24	ARM
I^2C 0 RX	9	Synopsys	Quad SPI Flash RX	25	ARM
I^2C 1 TX	10	Synopsys	STM	26	ARM
I^2C 1 RX	11	Synopsys	Reserved	27	—
I^2C 2 TX (EMAC)	12	Synopsys	UART 0 TX	28	Synopsys
I^2C 2 RX (EMAC)	13	Synopsys	UART 0 RX	29	Synopsys
I^2C 3 TX (EMAC)	14	Synopsys	UART 1 TX	30	Synopsys
I^2C 3 RX (EMAC)	15	Synopsys	UART 1 RX	31	Synopsys

Request interface numbers 4 through 7 are multiplexed between the CAN controllers and soft logic implemented in the FPGA fabric. The switching between the CAN controller and FPGA interfaces is controlled by the system manager.

For more information about controlling request interface numbers 4 through 7 refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Request Acceptance Capability

The DMAC is able to accept one active request for each peripheral request interface. An active request is where the DMAC has not started the requested AXI data transfers.

Peripheral Length Management

The peripheral request interface enables a peripheral to control the quantity of data that a DMA cycle contains, without the DMAC being aware of how many data transfers it contains. The peripheral controls the DMA cycle in one of the following ways:

- Selects a single transfer
- Selects a burst transfer
- Notifies the DMAC when it commences the final request in the current series

When the DMAC executes a `DMAWFP` periph instruction, it halts execution of the thread and waits for the peripheral to send a request. When the peripheral sends the request, the DMAC sets the state of the request flags depending on the state of the following signals:

- `drtype_<x>[1:0]`—The DMAC sets the state of the `request_type` flag:
  - `drtype_<x>[1:0]`=b00—`request_type<x>` = Single.
  - `drtype_<x>[1:0]`=b01—`request_type<x>` = Burst.
- `drlast_<x>`—The DMAC sets the state of the `request_last` flag:
  - `drlast_<x>`=0—`request_last<x>` = 0.
  - `drlast_<x>`=1—`request_last<x>` = 1.

If the DMAC executes a `DMAWFP` single or `DMAWFP` burst instruction then the DMAC sets:

- The `request_type<x>` flag to Single or Burst, respectively
- The `request_last<x>` flag to 0.

`DMALPFE` is an assembler directive which forces the associated `DMALPEND` instruction to have its `nf` bit set to 0. This creates a program loop that does not use a loop counter to terminate the loop.

The DMAC exits the loop when the `request_last` flag is set to 1.

The DMAC conditionally executes the following instructions, depending on the state of the `request_type` and `request_last` flags:

- `DMALD`, `DMAST`, `DMALPEND`

  When these instructions use the optional B | S suffix then the DMAC executes a `DMANOP` if the `request_type` flag does not match.

- DMALDP<B|S>, DMASTP<B|S>

  The DMAC executes a DMANOP if the request_type<*x*> flag does not match the B|S

- DMALPEND

  When the nf bit is 0, the DMAC executes a DMANOP if the request_last flag is set.

Use the DMALDB, DMALDPB, DMASTB and DMASTPB instructions if you require the DMAC to issue a burst transfer when the DMAC receives a burst request. The values in the CCR*n* register control the amount of data that the DMAC transfers.

Use the DMALDS, DMALDPS, DMASTS and DMASTPS instructions if you require the DMAC to issue a single transfer when the DMAC receives a single request. The DMAC ignores the value of the src_burst_len and dst_burst_len fields in the CCRn register and sets the arlen[3:0] or awlen[3:0] buses to 0x0.

DMAC Length Management

DMAC length management is when the DMAC controls the total amount of data to transfer. The peripheral uses the peripheral request interface to notify the DMAC when it requires the DMAC to transfer data to or from the peripheral. The DMA channel thread controls how the DMAC responds to the peripheral requests.

The following constraints apply to DMAC length management:

- The total quantity of data for all the single requests from a peripheral must be less than the quantity of data for a burst request for that peripheral.

  ☞ The CCR*n* register controls how much data is transferred for a burst request and a single request. Altera recommends that you do not update a CCR*n* register when a transfer is in progress for channel *n*.

- When the peripheral sends a burst request then the peripheral must not send a single request until the DMAC acknowledges that the burst request is complete.

Use the DMAWFP single instruction when you require the program thread to halt execution until the peripheral request interface receives any request type. If the head entry in the request FIFO buffer is of request type:

- Single—The DMAC pops the entry from the FIFO buffer and continues program execution.
- Burst—The DMAC leaves the entry in the FIFO buffer and continues program execution.

  ☞ The burst request entry remains in the request FIFO buffer until the DMAC executes a DMAWFP burst instruction or a DMAFLUSHP instruction.

Use the DMAWFP burst instruction when you require the program thread to halt execution until the peripheral request interface receives a burst request. If the head entry in the request FIFO buffer is of request type:

- Single—The DMAC removes the entry from the FIFO buffer and program execution remains
- Burst—The DMAC pops the entry from the FIFO buffer and continues program execution.

Use the DMALDP instruction when you require the DMAC to send an acknowledgement to the peripheral when it completes the AXI read transfers. Similarly, use the DMASTP instruction when you require the DMAC to send an acknowledgement to the peripheral when it completes the AXI write transfers. The DMAC uses the acknowledge bus to signal a transfer acknowledgement to peripheral <x>.

The DMAC sends an acknowledgement for a read transaction when the rvalid and rlast signals are high and for a write transaction when bvalid signal is high. The DMAC might send an acknowledgement to the peripheral while the transfer of write data to the end destination is still in progress.

Use the DMAFLUSHP instruction to reset the request FIFO buffer for the peripheral request interface. After the DMAC executes DMAFLUSHP, it ignores peripheral requests until the peripheral acknowledges the flush request. This enables the DMAC and peripheral to synchronize with each other.

Limitations

The peripherals connected to the DMA peripheral request interface use one of the following protocols:

- ARM
- Synopsys
- Bosch

For a peripheral using the ARM protocol, the only logic between the DMA and the peripheral is clock crossing logic. For other protocols, clock crossing and protocol adaptation logic is located between the DMA and peripheral. The following sections discuss the limitations of the peripheral handshake interface for peripherals using the Synopsys or Bosch protocols.

Burst Only Request

The Bosch protocol used by the CAN controllers only supports burst requests. As a result any time either CAN controller issues the peripheral burst request the DMA controller receives a burst and single request.

No Flush Support

For peripherals that use the Bosch or Synopsys peripheral request protocols the flush command is not supported. Issuing a flush to any peripheral interface supporting the Bosch or Synopsys protocols is dropped before reaching the peripheral.

No Acknowledge Type

For the Bosch peripheral request interface protocol, there is no acknowledge available. Issuing an acknowledge to either CAN controller as a result is dropped before reaching the peripheral.

For the Synopsys peripheral request interface protocol, there is no acknowledge type available. As a result peripherals using the Synopsys protocol cannot distinguish between acknowledges to burst and single transfers.

Using Events and Interrupts

The DMAC can support eight events and interrupts. The INTEN register is used to control if each event-interrupt resource is either an event or an interrupt.

When the DMAC executes a DMASEV instruction it modifies the event-interrupt resource that you specify. The INTEN register sets the event-interrupt resource to function as an:

- **Event**—The DMAC generates an event for the specified event-interrupt resource. When the DMAC executes a DMAWFE instruction for the same event-interrupt resource then it clears the event.
- **Interrupt**—The DMAC sets the irq<*event_num*> signal high, where <*event_num*> is the number of the specified event resource. To clear the interrupt you must write to the INTCLR register.

Using an Event to Restart DMA Channels

When you program the INTEN register to generate an event, you can use the DMASEV and DMAWFE instructions to restart one or more DMA channels.

DMAC executes DMAWFE before DMASEV

To restart a single DMA channel:

1. The first DMA channel executes DMAWFE and then stalls while it waits for the event to occur.
2. The other DMA channel executes DMASEV using the same event number. This generates an event, and the first DMA channel restarts. The DMAC clears the event, one clock cycle after it executes DMASEV.

You can program multiple channels to wait for the same event. For example, if four DMA channels have all executed DMAWFE for event 2, then when another DMA channel executes DMASEV for event 2, the four DMA channels all restart at the same time. The DMAC clears the event, one clock cycle after it executes DMASEV.

DMAC executes DMASEV before DMAWFE

If the DMAC executes DMASEV before another channel executes DMAWFE then the event remains pending until the DMAC executes DMAWFE. When the DMAC executes DMAWFE it halts execution for one aclk clock cycle, clears the event and then continues execution of the channel thread.

For example, if the DMAC executes DMASEV 6 and none of the other threads have executed DMAWFE 6 then the event remains pending. If the DMAC executes DMAWFE 6 instruction for channel 4 and then executes DMAWFE 6 instruction for channel 3, then:

1. The DMAC halts execution of the channel 4 thread for one aclk clock.
2. The DMAC clears event 6.
3. The DMAC resumes execution of the channel 4 thread.
4. The DMAC halts execution of the channel 3 thread and the thread stalls while it waits for the next occurrence of event 6.

Interrupting the MPU Subsystem

The DMAC provides the `irq[x]` signals for use as active-high level-sensitive interrupts to the MPU subsystem. When you program the `INTEN` register to generate an interrupt, after the DMAC executes `DMASEV`, it sets the corresponding `irq[x]` signal high.

The MPU subsystem can clear the interrupt by writing to the `INTCLR` register.

☞ Executing `DMAWFE` does not clear an interrupt.

If you use the `DMASEV` instruction to notify a microprocessor when the DMAC completes a `DMALD` or `DMAST` instruction then Altera recommends that you insert a memory barrier instruction before the `DMASEV`. Otherwise the DMAC might signal an interrupt before the AXI transfers complete. Refer to Example 16–1.

Example 16–1. Memory Barrier Instruction

```
DMALD
DMAST
# Issue a write memory barrier
# Wait for the AXI write transfer to complete before the DMAC
# can send an interrupt
DMAWMB
# The DMAC sends the interrupt
DMASEV
```

Aborts

This section describes:

- *Abort Types*
- *Abort Sources*
- Watchdog Abort
- Abort Handling

Abort Types

An abort can be classified as either precise or imprecise, depending on whether the DMAC provides an abort handler with the precise state of the DMAC when the abort occurs. If an abort is:

- **Precise** The DMAC updates the `PC` register with the address of the instruction that created the abort.
- **Imprecise** The `PC` register might contain the address of an instruction which did not cause the abort to occur.

Abort Sources

The DMAC signals a precise abort under the following conditions:

- A DMA channel thread in the Non-secure state attempts to program its `CCR`*n* register and generate a secure AXI transaction.

- A DMA channel thread in the Non-secure state executes DMAWFE or DMASEV for an event that is set as secure. The boot_irq_ns memory-mapped control signals initialize the security state for an event.

  For each event, the INTEN register controls if the DMAC generates an event or signals an interrupt.

- A DMA channel thread attempts to execute DMAST but the DMAC calculates that when it eventually performs the store, the MFIFO buffer contains insufficient data to enable it to complete the store.
- A DMA channel thread in the Non-secure state executes DMAWFP, DMALDP, DMASTP, or DMAFLUSHP for a peripheral request interface that is set as secure. The boot_periph_ns memory-mapped control signals initialize the security state for a peripheral request interface.
- A DMA manager thread in the Non-secure state executes DMAGO to attempt to start a secure DMA channel thread.
- The DMAC receives an ERROR response on the AXI master interface when it performs an instruction fetch.
- A thread executes an undefined instruction.
- A thread executes an instruction with an operand that is invalid for the configuration of the DMAC.

When the DMAC signals a precise abort, the instruction that triggers the abort is not executed. Instead, the DMAC executes a DMANOP.

The DMAC signals an imprecise abort under the following conditions:

- The DMAC receives an ERROR response on the AXI master interface when it performs a data load
- The DMAC receives an ERROR response on the AXI master interface when it performs a data store
- A DMA channel thread executes DMALD or DMAST, and the MFIFO buffer is too small to hold the required amount of data
- A DMA channel thread executes DMAST but the thread has not executed sufficient DMALD instructions
- A DMA channel thread locks up because of resource starvation, and this causes the internal watchdog timer to time out.

Watchdog Abort

The DMAC can lock up if one or more DMA channel programs are running and the MFIFO buffer is too small to satisfy the storage requirements of the DMA programs.

The DMAC contains logic to prevent it from remaining in a state where it is unable to complete a DMA transfer.

The DMAC detects a lock up when all of the following conditions occur:

- Load queue is empty

- Store queue is empty
- All of the running channels are prevented from executing a DMALD instruction either because the MFIFO buffer does not have sufficient free space or another channel owns the load-lock.

When the DMAC detects a lockup it signals an interrupt and can also abort the contributing channels. The DMAC behavior depends on the state of the wd_irq_only bit in the WD register.

If:

- wd_irq_only=0—The DMAC aborts all of the contributing DMA channels and sets the irq_abort signal high.
- wd_irq_only=1—The DMAC sets the irq_abort signal high.

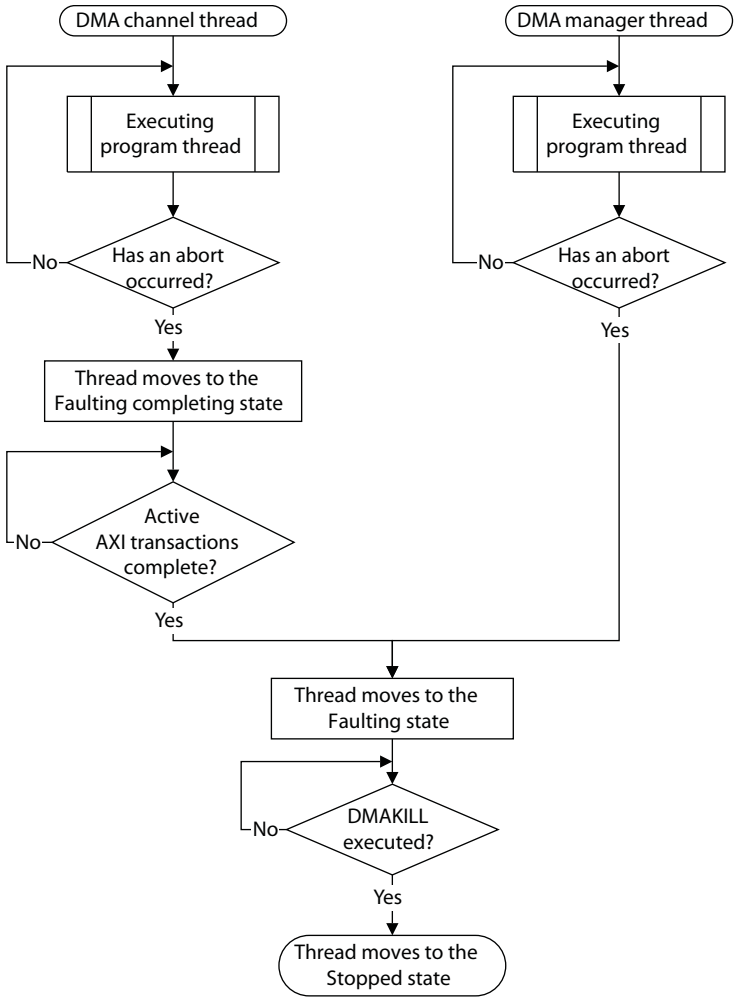
For more information, refer to "Resource Sharing Between DMA Channels" on page 16–26.

Abort Handling

The architecture of the DMAC is not designed to recover from an abort and you must therefore use an external agent, such as a microprocessor, to terminate a thread when an abort occurs.

Figure 16–4 shows the operating states for the DMA channel and DMA manager threads after an abort occurs.

Figure 16–4. Abort Process



After an abort occurs, the action the DMAC takes depends on the thread type:

- DMA channel thread—The thread immediately moves to the Faulting completing state. In this state, the DMAC:
  - Sets the `irq_abort` signal high
  - Stops executing instructions for the DMA channel
  - Invalidates all cache entries for the DMA channel updates the `CPC`*n* register to contain the address of the aborted instruction provided that the abort is precise
  - Does not generate AXI accesses for any instructions remaining in the read queue and write queue
  - Permits currently active AXI transactions to complete.

  After the transactions for the DMA channel complete, the thread moves to the Faulting state.

- DMA manager thread—The thread immediately moves to the Faulting state and the DMAC sets the `irq_abort` signal high.

  The external agent can respond to the assertion of the `irq_abort` signal by:

  - Reading the status of the `FSRD` register to determine if the DMA manager is Faulting. In the Faulting state, the `FSRD` register provides the cause of the abort.
  - Reading the status of the `FSRC` register to determine if a DMA channel is Faulting. In the Faulting state, the `FSRC` register provides the cause of the abort.

  To enable a thread in the Faulting state to move to the Stopped state, the external agent must:

  - Program the `DBGINST0` register with the encoding for the `DMAKILL` instruction.
  - Write to the `DBGCMD` register.

  ☞ If the aborted thread is secure, you must use the secure slave interface to update these

After a thread in the Faulting state executes `DMAKILL`, it moves to the Stopped state.

Security Usage

When the DMAC exits from reset, the status of the configuration signals configures the security for:

- DMA manager thread—The `DNS` bit in the DSR register returns the security state of the DMA manager thread.
- Events and interrupts—The `INS` bit in the `CR3` register returns the security state of the event-interrupt resources.
- Peripheral request interfaces—The `PNS` bit in the `CR4` register returns the security state of these interfaces.

Additionally, each DMA channel thread contains a dynamic non-secure bit, `CNS`, that is valid when the channel is not in the Stopped state.

DMA Manager Thread in Secure State

If the `DNS` bit is 0, the DMA manager thread operates in the Secure state and it only performs secure instruction fetches. When a DMA manager thread in the Secure state processes:

- `DMAGO`—The DMAC uses the status of the `ns` bit, to set the security state of the DMA channel thread by writing to the `CNS` bit for that channel.
- `DMAWFE`—The DMAC halts execution of the thread until the event occurs. When the event occurs, the DMAC continues execution of the thread, irrespective of the security state of the corresponding `INS` bit.
- `DMASEV`—The DMAC sets the corresponding bit in the INT_EVENT_RIS register, irrespective of the security state of the corresponding `INS` bit.

DMA Manager Thread in Non-Secure State

If the DNS bit is 1, the DMA manager thread operates in the Non-secure state, and it only performs non-secure instruction fetches. When a DMA manager thread in the Non-secure state processes:

- DMAGO—The DMAC uses the status of the ns bit, to control if it starts a DMA channel thread. If:
  - ns = 0—The DMAC does not start a DMA channel thread and instead it:
    - Executes a NOP.
    - Sets the FSRD register.
    - Sets the dmago_err bit in the FTRD register.
    - Moves the DMA manager to the Faulting state.
  - ns = 1—The DMAC starts a DMA channel thread in the Non-secure state and programs the CNS bit to be non-secure.
- DMAWFE—The DMAC uses the status of the corresponding INS bit, in the CR3 register, to control if it waits for the event. If:
  - INS = 0—The event is in the Secure state. The DMAC:
    - Executes a NOP.
    - Sets the FSRD register.
    - Sets the mgr_evnt_err bit in the FTRD register.
    - Moves the DMA manager to the Faulting state.
  - INS = 1—The event is in the Non-secure state. The DMAC halts execution of the thread and waits for the event to occur.
- DMASEV—The DMAC uses the status of the corresponding INS bit, in the CR3 register, to control if it creates the event interrupt. If:
  - INS = 0—The event-interrupt resource is in the Secure state. The DMAC:
    - Executes a NOP.
    - Sets the FSRD register.
    - Sets the mgr_evnt_err bit in the FTRD register.
    - Moves the DMA manager to the Faulting state.
  - INS = 1—The event-interrupt resource is in the Non-secure state. The DMAC creates the event interrupt.

DMA Channel Thread in Secure State

When the CNS bit is 0, the DMA channel thread is programmed to operate in the Secure state and it only performs secure instruction fetches.

When a DMA channel thread in the Secure state processes the following instructions:

- DMAWFE—The DMAC halts execution of the thread until the event occurs. When the event occurs, the DMAC continues execution of the thread, irrespective of the security state of the corresponding INS bit, in the CR3 register.

- DMASEV—The DMAC creates the event interrupt, irrespective of the security state of the corresponding INS bit, in the CR3 register.
- DMAWFP—The DMAC halts execution of the thread until the peripheral signals a DMA request. When this occurs, the DMAC continues execution of the thread, irrespective of the security state of the corresponding PNS bit, in the CR4 register.
- DMALDP and DMASTP—The DMAC sends a message to the peripheral to communicate that data transfer is complete, irrespective of the security state of the corresponding PNS bit, in the CR4 register.
- DMAFLUSHP—The DMAC clears the state of the peripheral and sends a message to the peripheral to resend its level status, irrespective of the security state of the corresponding PNS bit, in the CR4 register.

When a DMA channel thread is in the Secure state, it enables the DMAC to perform secure and non-secure AXI accesses.

DMA Channel Thread in Non-Secure State

When the CNS bit is 1, the DMA channel thread is programmed to operate in the Non-secure state and it only performs non-secure instruction fetches.

When a DMA channel thread in the Non-secure state processes the following instructions:

- DMAWFE—The DMAC uses the status of the corresponding INS bit, in the CR3 register, to control if it waits for the event. If:
  - INS = 0—The event is in the Secure state. The DMAC:
    - Executes a NOP.
    - Sets the appropriate bit in the FSRC register that corresponds to the DMA channel number.
    - Sets the ch_evnt_err bit in the FTR*n* register.
    - Moves the DMA channel to the Faulting completing state.
  - INS = 1—The event is in the Non-secure state. The DMAC halts execution of the thread and waits for the event to occur.
- DMASEV—The DMAC uses the status of the corresponding INS bit, in the CR3 register, to control if it creates the event. If:
  - INS = 0—The event-interrupt resource is in the Secure state. The DMAC:
    - Executes a NOP.
    - Sets the appropriate bit in the FSRC register that corresponds to the DMA channel number.
    - Sets the ch_evnt_err bit in the FTR*n* register.
    - Moves the DMA channel to the Faulting completing state.
  - INS = 1—The event-interrupt resource is in the Non-secure state. The DMAC creates the event interrupt.

- DMAWFP—The DMAC uses the status of the corresponding PNS bit, in the CR4 register, to control if it waits for the peripheral to signal a request. If:
  - PNS = 0—The peripheral is in the Secure state. The DMAC:
    - Executes a NOP.
    - Sets the appropriate bit in the FSRC register that corresponds to the DMA channel number.
    - Sets the ch_periph_err bit in the FTR*n* register.
    - Moves the DMA channel to the Faulting completing state.
  - PNS = 1—The peripheral is in the Non-secure state. The DMAC halts execution of the thread and waits for the peripheral to signal a request.
- DMALDP and DMASTP—The DMAC uses the status of the corresponding PNS bit, in the CR4 register, to control if it sends an acknowledgement to the peripheral. If:
  - PNS = 0—The peripheral is in the Secure state. The DMAC:
    - Executes a NOP.
    - Sets the appropriate bit in the FSRC register that corresponds to the DMA channel number.
    - Sets the ch_periph_err bit in the FTR*n* register.
    - Moves the DMA channel to the Faulting completing state.
  - PNS = 1—The peripheral is in the Non-secure state. The DMAC sends a message to the peripheral to communicate when the data transfer is complete.
- DMAFLUSHP—The DMAC uses the status of the corresponding PNS bit, in the CR4 register, to control if it sends a flush request to the peripheral. If:
  - PNS = 0—The peripheral is in the Secure state. The DMAC:
    - Executes a NOP.
    - Sets the appropriate bit in the FSRC register that corresponds to the DMA channel number.
    - Sets the ch_periph_err bit in the FTR*n* register.
    - Moves the DMA channel to the Faulting completing state.
  - PNS = 1—The peripheral is in the Non-secure state. The DMAC clears the state of the peripheral and sends a message to the peripheral to resend its level status.

When a DMA channel thread is in the Non-secure state, and a DMAMOV CCR instruction attempts to program the channel to perform a secure AXI transaction, the DMAC:

1. Executes a DMANOP.
2. Sets the appropriate bit in the FSRC register that corresponds to the DMA channel number.
3. Sets the ch_rdwr_err bit in the FTR*n* register.
4. Moves the DMA channel thread to the Faulting completing state.

Constraints and Limitations of Use

This section describes constraints and use limitations.

DMA Channel Arbitration

The DMAC uses a round-robin scheme to service the active DMA channels. To ensure that the DMAC continues to service the DMA manager, it always services the DMA manager prior to servicing the next DMA channel.

It is not possible to alter the arbitration process of the DMAC.

DMA Channel Prioritization

The DMAC responds to all active DMA channels with equal priority. It is not possible to increase the priority of a DMA channel over any other DMA channels.

Instruction Cache Latency

When a cache miss occurs, the latency to service the request is mainly dependent on the read latency of the memory containing the DMA code. The latency that the DMAC adds is minimal.

AXI Data Transfer Size

The DMAC can only perform data accesses up to 64 bits in width. If you program the `src_burst_size` or `dst_burst_size` fields to be larger, the DMAC signals a precise abort. Refer to *"Abort Sources" on page 16–16* for more information.

AXI Bursts Crossing 4 KB Boundaries

The AXI specification does not permit AXI bursts to cross 4 KB address boundaries. If you program the DMAC with a combination of burst start address, size, and length that would cause a single burst to cross a 4 KB address boundary, then the DMAC instead generates a pair of bursts with a combined length equal to that specified. This operation is transparent to the DMAC channel thread program so that, for example, the DMAC responds to a single `DMALD` instruction by generating the appropriate pair of AXI read bursts.

AXI Burst Types

You can program the DMAC to generate only fixed-address or incrementing-address burst types for data accesses. It does not generate wrapping-address bursts for data accesses or for instruction fetches.

AXI Write Addresses

The DMAC can issue up to eight outstanding write addresses. The DMAC does not issue a write address until it has read in all of the data bytes required to fulfill that write transaction.

AXI Write Data Interleaving

The DMAC does not generate interleaved write data. All write data beats for one write transaction are output before any write data beat for the next write transaction.

Programming Restrictions

The following sections describe restrictions that apply when programming the DMAC.

Fixed Unaligned Bursts

The DMAC does not support fixed unaligned bursts. If you program the following conditions, the DMAC treats this as a programming error:

- Unaligned read
  - `src_inc` field is 0 in the `CCR`*n* register.
  - The `SAR`*n* register contains an address that is not aligned to the size of data that the `src_burst_size` field contains.
- **Unaligned write**
  - `dst_inc` field is 0 in the `CCR`*n* register.
  - The `DAR`*n* register contains an address that is not aligned to the size of data that the `dst_burst_size` field contains.

Endian Swap Size Restrictions

If you program the `endian_swap_size` field in the `CCR`*n* register, to enable a DMA channel to perform an endian swap, then you must set the corresponding `SAR`*n* register and the corresponding `DAR`*n* register to contain an address that is aligned to the size that the `endian_swap_size` field specifies before executing any `DMALD` or `DMAST` instructions.

If you update any of `endian_swap_size`, `SAR`*n*, or `DAR`*n*, for example, using a `DMAADDH SAR` instruction, then you must ensure that the `SAR`*n* and `DAR`*n* registers contain an address aligned to the size that the `endian_swap_size` field specifies before executing any additional `DMALD` or `DMAST` instructions.

If you program the `src_inc` field in the `CCR`*n* register to use a fixed address, you must program the `src_burst_size` field to select a burst size that is greater than or equal to the value that the `endian_swap_size` field specifies. Similarly, if you program the `dst_inc` field to select a fixed destination address, you must program the `dst_burst_size` field to select a burst size that is greater than or equal to the value that the `endian_swap_size` field specifies.

If you program the `dst_inc` field in the `CCR`*n* register to use an incrementing address, you must program the `CCR`*n* register so that dst_burst_len times `dst_burst_size` is a multiple of `endian_swap_size`. For example, if `endian_swap_size` = 2, 32-bit, and `dst_burst_size` = 1, 2 bytes per beat, then you can program dst_burst_len = 1, 3, 5, ..., 15, that is 2, 4, 6, ..., 16 transfers.

Updating DMA Channel Control Registers During a DMA Cycle

Prior to the DMAC executing a sequence of `DMALD` and `DMAST` instructions, the values you program in to the `CCR`*n* register, `SAR`*n* register, and `DAR`*n* register control the data byte lane manipulation that the DMAC performs when it transfers the data from the source address to the destination address.

You can update these registers during a DMA cycle but if you change certain register fields then it can cause the DMAC to discard data. The following sections describe the register fields that might have a detrimental impact on a data transfer.

Updates that affect the destination address

If you use a DMAMOV instruction to update the DAR*n* register or CCR*n* register part way through a DMA cycle then this might cause a discontinuity in the destination data stream.

A discontinuity occurs if you change any of the following:

- endian_swap_size field.
- dst_inc bit.
- dst_burst_size field when dst_inc = 0, that is, fixed-address burst.
- DAR*n* register so that it modifies the destination byte lane alignment. Because the bus width is 64 bits, you change bits [2:0] in the DAR*n* register.

When a discontinuity in the destination data stream occurs, the DMAC:

1. Halts execution of the DMA channel thread.
2. Completes all outstanding read and write operations for the channel. That is, as if the DMAC were executing DMARMB and DMAWMB.
3. Discards any residual MFIFO buffer data for the channel.
4. Resumes execution of the DMA channel thread.

Updates that affect the source address

If you use a DMAMOV instruction to update the SAR*n* register or CCR*n* register part way through a DMA cycle then this might cause a discontinuity in the source data stream.

A discontinuity occurs if you change any of the following:

- src_inc bit.
- src_burst_size field.
- SAR*n* register so that it modifies the source byte lane alignment. Because the bus width is 64 bits, you change bits [2:0] in the SAR*n* register.

When a discontinuity in the source data stream occurs, the DMAC:

1. Halts execution of the DMA channel thread.
2. Completes all outstanding read operations for the channel. That is, as if the DMAC were executing DMARMB
3. Resumes execution of the DMA channel thread. No data is discarded from the MFIFO buffer.

Resource Sharing Between DMA Channels

DMA channel programs share the MFIFO buffer data storage resource. You must not start a set of concurrently running DMA channel programs with a resource requirement that exceeds 512, the size of the MFIFO buffer. If you exceed this limit then the DMAC might lock up and generate a Watchdog abort, refer to "Watchdog Abort" on page 16–17.

The DMAC includes a mechanism called the *load-lock* to ensure that the shared MFIFO buffer resource is used correctly. The load-lock is either owned by one channel, or it is free. The channel that owns the load-lock can execute `DMALD` instructions successfully. A channel that does not own the load-lock pauses at a `DMALD` instruction until it takes ownership of the load-lock.

A channel claims ownership of the load lock when:

- It executes a `DMALD` or `DMALDP` instruction
- No other channel currently owns the load-lock.

A channel releases ownership of the load-lock when any of the following occur:

- It executes a `DMAST`, `DMASTP`, or `DMASTZ`
- It reaches a barrier, that is, it executes `DMARMB` or `DMAWMB`
- It waits, that is, it executes `DMAWFP` or `DMAWFE`
- It terminates normally, that is, it executes `DMAEND`
- It aborts for any reason, including `DMAKILL`.

The MFIFO buffer resource usage of a DMA channel program is measured in MFIFO buffer entries, and rises and falls as the program proceeds. The MFIFO buffer resource requirement of a DMA channel program is described using a *static requirement* and a *dynamic requirement* which are affected by the load-lock mechanism.

The static requirement is defined to be the maximum number of MFIFO buffer entries that a channel is currently using before that channel does one of the following:

- Executes a WFP or WFE instruction
- Claims ownership of the load-lock.

The dynamic requirement is defined to be the difference between the static requirement and the maximum number of MFIFO buffer entries that a channel program uses at any time during its

To calculate the total MFIFO buffer requirement, add the largest dynamic requirement to the sum of all the static requirements.

To avoid DMAC lockup, the total MFIFO buffer requirement of the set of channel programs must be equal to or less than 512, the MFIFO buffer depth.

For more information, refer to *"MFIFO Buffer Usage Overview" on page 16–47*.

DMA Controller Programming Model

This section describes the instruction set of the DMAC.

Instruction Syntax Conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

- < >

  Any item bracketed by < and > is mandatory. A description of the item and of how it is encoded in the instruction is supplied by subsequent text.

- [ ]

  Any item bracketed by [and] is optional. A description of the item and of how its presence or absence is encoded in the instruction is supplied by subsequent

- **Spaces**

  Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.

Instruction Set Summary

The DMAC instructions:

- Use a DMA prefix, to provide a unique name-space
- Have 8-bit opcodes that might use a variable data payload of 0, 8, 16, or 32 bits
- Use suffixes that are consistent.

Table 16–2 shows a summary of the instruction syntax.

Table 16–2. Instruction Syntax Summary

Mnemonic	Instruction	DMA Manager Usage	DMA Channel Usage	Description
DMAADDH	Add Halfword	No	Yes	Refer to "DMAADDH" on page 16–29.
DMAADNH	Add Negative Halfword	No	Yes	Refer to "DMAADNH" on page 16–29.
DMAEND	End	Yes	Yes	Refer to "DMAEND" on page 16–30.
DMAFLUSHP	Flush and Notify Peripheral	No	Yes	Refer to "DMAFLUSHP" on page 16–31.
DMAGO	Go	Yes	No	Refer to "DMAGO" on page 16–31.
DMAKILL	Kill	Yes	Yes	Refer to "DMAKILL" on page 16–32.
DMALD	Load	No	Yes	Refer to "DMALD[S | B]" on page 16–33.
DMALDP	Load and Notify Peripheral	No	Yes	Refer to "DMALDP<S | B>" on page 16–34.
DMALP	Loop	No	Yes	Refer to "DMALP" on page 16–35.
DMALPEND	Loop End	No	Yes	Refer to "DMALPEND[S | B]" on page 16–36.
DMALPFE	Loop Forever	No	Yes	Refer to "DMALPFE" on page 16–38.
DMAMOV	Move	No	Yes	Refer to "DMAMOV" on page 16–38.
DMANOP	No Operation	Yes	Yes	Refer to "DMANOP" on page 16–39.
DMARMB	Read Memory Barrier	No	Yes	Refer to "DMARMB" on page 16–39.
DMASEV	Send Event	Yes	Yes	Refer to "DMASEV" on page 16–40.
DMAST	Store	No	Yes	Refer to "DMAST[S | B]" on page 16–40.
DMASTP	Store and Notify Peripheral	No	Yes	Refer to "DMASTP<S | B>" on page 16–41.
DMASTZ	Store Zero	No	Yes	Refer to "DMASTZ" on page 16–42.
DMAWFE	Wait For Event	Yes	Yes	Refer to "DMAWFE" on page 16–43.
DMAWFP	Wait For Peripheral	No	Yes	Refer to "DMAWFP" on page 16–43.
DMAWMB	Write Memory Barrier	No	Yes	Refer to "DMAWMB" on page 16–44.

Instructions

The following sections describe the instructions that a DMAC can execute.

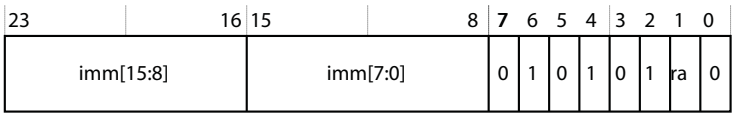
DMAADDH

Add Halfword adds an immediate 16-bit value to the SAR*n* register or DAR*n* register, for the DMA channel thread. This enables the DMAC to support 2D DMA operations.

The immediate unsigned 16-bit value is zero-extended before the DMAC adds it to the address, using 32-bit addition. The DMAC discards the carry bit so that addresses wrap from 0xFFFFFFFF to 0x00000000.

Figure 16–5 shows the instruction encoding.

Figure 16–5. DMAADDH Encoding



Assembler syntax

```
DMAADDH <address_register>, <16-bit immediate>
```

where:

`<address_register>` Selects the address register to use. It must be either:

- `SAR` SAR*n* register and sets `ra` to 0
- `DAR` DAR*n* register and sets `ra` to 1

`<16-bit immediate>` The immediate value to be added to the `<address_register>`.

Operation

You can only use this instruction in a DMA channel thread.

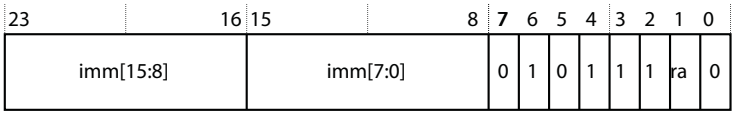
DMAADNH

Add Negative Halfword adds an immediate negative 16-bit value to the SAR*n* register or DAR*n* register, for the DMA channel thread. This enables the DMAC to support 2D DMA operations, or reading or writing an area of memory in a different order to naturally incrementing addresses.

The immediate unsigned 16-bit value is one-extended to 32 bits, to create a value that is the two's complement representation of a negative number between -65536 and -1, before the DMAC adds it to the address using 32-bit addition. The DMAC discards the carry bit so that addresses wrap from 0xFFFFFFFF to 0x00000000. The net effect is to subtract between 65536 and 1 from the current value in the source or destination address register.

Figure 16–6 shows the instruction encoding.

Figure 16–6. DMAADNH Encoding



Assembler syntax

```
DMAADNH <address_register>, <16-bit immediate>
```

where:

`<address_register>` Selects the address register to use. It must be either:

SAR SAR*n* register and sets `ra` to 0

DAR DAR*n* register and sets `ra` to 1

`<16-bit immediate>` The immediate value to be added to the `<address_register>`.

☞ You should specify the 16-bit immediate as the number that is to be represented in the instruction encoding. For example, `DMAADNH DAR`, 0xFFF0 causes the value 0xFFFFFFF0 to be added to the current value of the Destination Address register, effectively subtracting 16 from the DAR.

Operation

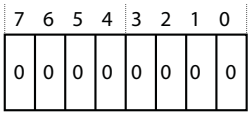
You can only use this instruction in a DMA channel thread.

DMAEND

End signals to the DMAC that the DMA sequence is complete. After all DMA transfers are complete for the DMA channel, the DMAC moves the channel to the Stopped state. It also flushes data from the MFIFO buffer and invalidates all cache entries for the thread.

Figure 16–7 shows the instruction encoding.

Figure 16–7. DMAEND Encoding



Assembler syntax

```
DMAEND
```

Operation

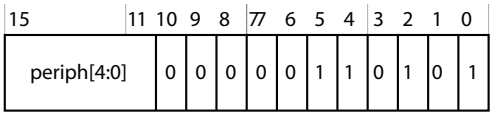
You can use the instruction with the DMA manager thread and the DMA channel thread.

DMAFLUSHP

Flush Peripheral clears the state in the DMAC that describes the contents of the peripheral and sends a message to the peripheral to resend its level status.

Figure 16–8 shows the instruction encoding.

Figure 16–8. DMAFLUSHP Encoding



Assembler syntax

```
DMAFLUSHP <peripheral>
```

where:

`<peripheral>` 5-bit immediate, value 0-31

Operation

You can only use this instruction in a DMA channel thread.

DMAGO

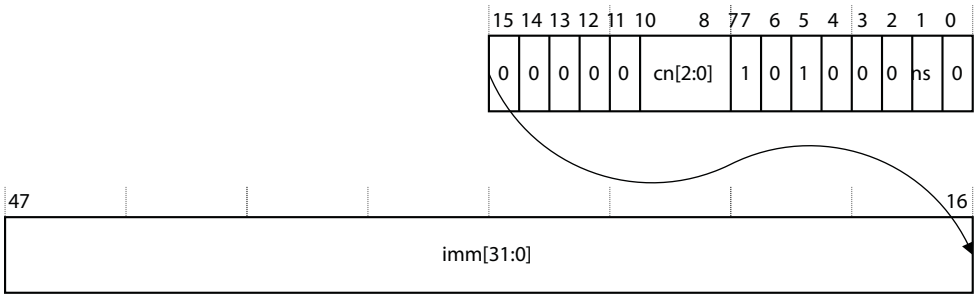
When the DMA manager executes Go for a DMA channel that is in the Stopped state, it performs the following steps on the DMA channel:

1. Moves a 32-bit immediate into the program counter
2. Sets its security state
3. Updates it to the Executing state.

If a DMA channel is not in the Stopped state when the DMA manager executes `DMAGO` then the DMAC does not execute `DMAGO` but instead it executes `DMANOP`.

Figure 16–9 shows the instruction encoding.

Figure 16–9. DMAGO Encoding



Assembler syntax

`DMAGO <channel_number>, <32-bit_immediate> [, ns]`

where:

`<channel_number>` Selects a DMA channel. It must be one of:

`C0` DMA channel 0

`C1` DMA channel 1

`C2` DMA channel 2

`C3` DMA channel 3

`C4` DMA channel 4

`C5` DMA channel 5

`C6` DMA channel 6

`C7` DMA channel 7

If you provide a channel number that is not available for your configuration of the DMAC, the DMA manager thread aborts.

`<32-bit_immediate>` The immediate value that is written to the `CPC`*n* register for the selected `<channel_number>`.

`[ns]`

- If `ns` is present, the DMA channel operates in the Non-secure state.
- Otherwise, the execution of the instruction depends on the security state of the DMA manager:

  DMA manager is in the Secure state—DMA channel operates in the Secure state.

  DMA manager is in the Non-secure state—The DMAC aborts.

Operation

You can only use this instruction with the DMA manager thread.

DMAKILL

Kill instructs the DMAC to immediately terminate execution of a thread. Depending on the thread type, the DMAC performs the following steps:

DMA Manager Thread

1. Invalidates all cache entries for the DMA manager.
2. Moves the DMA manager to the Stopped state.

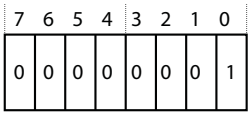
DMA Channel Thread

1. Moves the DMA channel to the Killing state.
2. Waits for AXI transactions, with an ID equal to the DMA channel number, to complete.
3. Invalidates all cache entries for the DMA channel.

4. Remove all entries in the MFIFO buffer for the DMA channel.
5. Remove all entries in the read buffer queue and write buffer queue for the DMA channel.
6. Moves the DMA channel to the Stopped state.

Figure 16–10 shows the instruction encoding.

Figure 16–10. DMAKILL Encoding



Assembler syntax

```
DMAKILL
```

Operation

You can use the instruction with the DMA manager thread and the DMA channel thread.

You must not use the `DMAKILL` instruction in DMA channel programs. To issue a `DMAKILL` instruction, use the `DBGINST0` register.

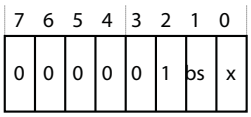
DMALD[S | B]

Load instructs the DMAC to perform a DMA load, using AXI transactions that the source address registers and channel control registers specify. It places the read data into the MFIFO buffer and tags it with the corresponding channel number. `DMALD` is an unconditional instruction but `DMALDS` and `DMALDB` are conditional on the state of the `request_type` flag. If the `src_inc` bit in the channel control registers is set to incrementing, the DMAC updates the source address registers after it executes `DMALD[S|B]`.

The DMAC sets the value of `request_type` when it executes a `DMAWFP` instruction. Refer to "DMAWFP" on page 16–43.

Figure 16–11 shows the instruction encoding.

Figure 16–11. DMALD[S|B] Encoding



Assembler syntax

```
DMALD[S|B]
```

where:

`[S]` If S is present, the assembler sets `bs` to 0 and `x` to 1. The instruction is conditional on the state of the `request_type` flag:

- request_type = Single

  The DMAC performs a DMALD instruction and it sets arlen[3:0]=0x0 so that the AXI read transaction length is one. The DMAC ignores the value of the src_burst_len field in the channel control registers.

- request_type = Burst

  The DMAC performs a DMANOP instruction. The DMAC increments the channel PC to the next instruction. No state change occurs.

[B] If B is present, the assembler sets bs to 1 and x to 1. The instruction is conditional on the state of the request_type flag:

- request_type = Single

  The DMAC performs a DMANOP instruction. The DMAC increments the channel PC to the next instruction. No state change occurs.

- request_type = Burst

  The DMAC performs a DMALD.

If you do not specify the S or B operand, the assembler sets bs to 0 and x to 0, and the DMAC always executes a DMA load.

Operation

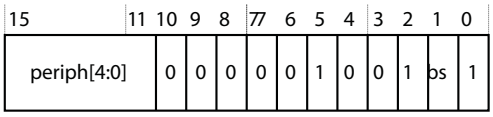
You can only use this instruction in a DMA channel thread. If you specify the S or B operand, execution of the instruction is conditional on the state of request_type matching that of the instruction.

DMALDP<S | B>

Load and notify Peripheral instructs the DMAC to perform a DMA load, using AXI transactions that source address registers and channel control registers specify. It places the read data into a FIFO buffer that is tagged with the corresponding channel number and after it receives the last data item, it sends an acknowledgement to the peripheral that the data transfer is complete. If the src_inc bit in the channel control registers is set to incrementing, the DMAC updates source address registers after it executes DMALDP<S|B>.

Figure 16–12 shows the instruction encoding.

Figure 16–12. DMALDP<S|B> Encoding



Assembler syntax

DMALDP<S|B> <peripheral>

where:

<S> When S is present, the assembler sets bs to 0. The instruction is conditional on the state of the request_type flag:

- `request_type` = Single

  The DMAC performs a `DMALDP` instruction and it sets `arlen[3:0]`=0x0 so that the AXI read transaction length is one. The DMAC ignores the value of the src_burst_len field in the channel control registers.

- `request_type` = Burst

  The DMAC performs a `DMANOP`.

`<B>` When B is present, the assembler sets `bs` to 1. The instruction is conditional on the state of the `request_type` flag:

- `request_type` = Single

  The DMAC performs a `DMANOP`.

- `request_type` = Burst

  The DMAC performs a load using a burst DMA transfer.

`<peripheral>` 5-bit immediate, value 0-31.

The DMAC sets the value of the `request_type` flag when it executes a `DMAWFP` instruction. Refer to "DMAWFP" on page 16–43.

Operation

You can only use this instruction in a DMA channel thread. Execution of the instruction is conditional on the state of the `request_type` flag matching that of the instruction.

DMALP

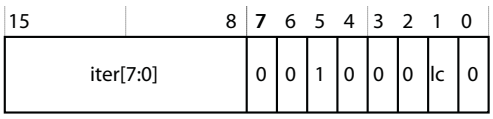
Loop instructs the DMAC to load an 8-bit value into the loop counter register you specify.

This instruction indicates the start of a section of instructions, and you set the end of the section using the `DMALPEND` instruction. Refer to "DMALPEND[S | B]" on page 16–36. The DMAC repeats the set of instructions that you insert between `DMALP` and `DMALPEND` until the value in the loop counter register reaches zero.

The DMAC saves the value of the PC for the instruction that follows `DMALP`. After the DMAC executes `DMALPEND`, and the loop counter register is not zero, this enables it to execute the first instruction in the loop.

Figure 16–13 shows the instruction encoding.

Figure 16–13. DMALP Encoding



Assembler syntax

`DMALP <loop_iterations>`

where:

`<loop_iterations>`

Specifies the number of loops to perform, range 1-256.

- The assembler determines the loop counter register to use and either:
  - Sets `lc` to 0, and the DMAC writes the value loop_iterations minus 1 to the loop counter 0 registers
  - Sets `lc` to 1, and the DMAC writes the value loop_iterations minus 1 to the loop counter 1 registers.

Operation

You can only use this instruction in a DMA channel thread.

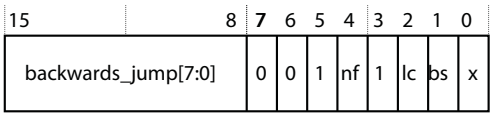
DMALPEND[S | B]

Loop End indicates the last instruction in the program loop but the behavior of the DMAC depends on whether `DMALP` or `DMALPFE` starts the loop. If a loop starts with:

- `DMALP` The loop has a defined loop count and `DMALPEND[S|B]` instructs the DMAC to read the value of the loop counter register. If a loop counter register returns:
  - Zero—The DMAC executes a `DMANOP` and therefore exits the loop.
  - Nonzero—The DMAC decrements the value in the loop counter register and updates the thread PC to contain the address of the first instruction in the program loop, that is, the instruction that follows the `DMALP`.
- `DMALPFE` The loop has an undefined loop count and the DMAC uses the state of the `request_last` flag to control when it exits the loop. If the `request_last` flag is:
  - 0—The DMAC updates the thread PC to contain the address of the first instruction in the program loop, that is, the instruction that follows the `DMALP`.
  - 1—The DMAC executes a `DMANOP` and therefore exits the loop.

Figure 16–14 shows the instruction encoding.

Figure 16–14. DMALPEND[S|B] Encoding



Assembler syntax

`DMALPEND[S|B]`

where:

`[S]` If S is present and the loop starts with `DMALP`, then the assembler sets `bs` to 0 and `x` to 1. The instruction is conditional on the state of the `request_type` flag:

- `request_type` = Single
  - The DMAC executes the `DMALPEND`.
- `request_type` = Burst
  - The DMAC performs a `DMANOP` and therefore exits the loop.

`[B]` If B is present and the loop starts with `DMALP`, then the assembler sets `bs` to 1 and `x` to 1. The instruction is conditional on the state of the `request_type` flag:

- `request_type` = Single
  - The DMAC performs a `DMANOP` and therefore exits the loop.
- `request_type` = Burst
  - The DMAC executes the `DMALPEND`

If you do not specify the S or B operand, the assembler sets `bs` to 0 and `x` to 0, and the DMAC always executes the `DMALPEND`.

You must not specify the S or B operand when a loop starts with `DMALPFE`. If you do, the assembler issues a warning message and sets `bs` to 0, `x` to 0, and `nf` to 1. In the same way as for `DMALPFE`, the DMAC uses the state of the `request_last` flag to control when it exits the loop.

The DMAC sets the value of the:

- `request_type` flag when it executes a `DMAWFP` instruction. Refer to "DMAWFP" on page 16–43.
- `request_last` flag to 1 when the corresponding peripheral issues the last request command through the peripheral request interface. For more information, refer to "Peripheral Length Management" on page 16–12.

To correctly assign the additional bits in the `DMALPEND` instruction, that Figure 16–14 shows, the assembler determines the values for:

`backwards_jump[7:0]` Sets the relative location of the first instruction in the program loop. The assembler calculates the value for `backwards_jump[7:0]` by subtracting the address of the first instruction in the loop from the address of the `DMALPEND`.

- `nf` sets it to:
  - 0 if `DMALPFE` started the program loop
  - 1 if `DMALP` started the program loop.
- `lc` sets it to:
  - 0 if the loop counter 0 registers contains the loop counter value
  - 1 if the loop counter 1 registers contains the loop counter value
  - 1 if `DMALPFE` started the program loop.

Operation

You can only use this instruction in a DMA channel thread. If you specify the S or B operand, execution of the instruction is conditional on the state of the `request_type` flag matching that of the instruction.

DMALPFE

The assembler uses Loop Forever to configure certain bits in DMALPEND. Refer to "DMALPEND[S | B]" on page 16–36.

When the assembler encounters DMALPFE, it does not create an instruction for the DMAC, but instead, it modifies the behavior of DMALPEND. The insertion of DMALPFE in program code identifies the start of the loop.

Assembler syntax

```
DMALPFE
```

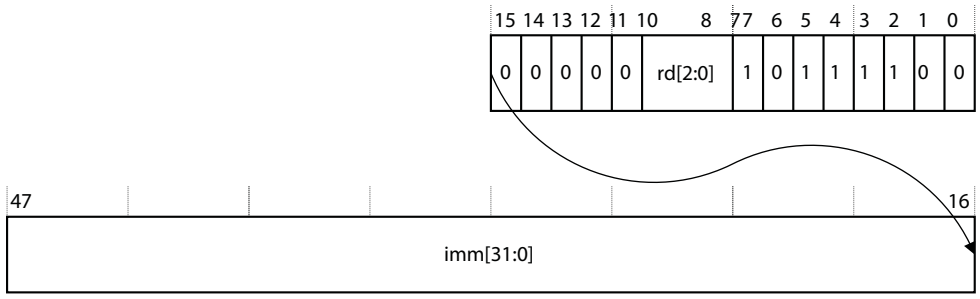
DMAMOV

Move instructs the DMAC to move a 32-bit immediate into the following registers:

- source address registers
- destination address registers
- channel control registers

Figure 16–15 shows the instruction encoding.

Figure 16–15. DMAMOV Encoding



Assembler syntax

```
DMAMOV <destination_register>, <32-bit_immediate>
```

where:

`<destination_register>`

The valid registers are:

- SAR—selects the source address registers and sets rd to b000
- CCR—selects the channel control registers and sets rd to b001
- DAR—selects the destination address registers and sets rd to b010

`<32-bit_immediate>`

A 32-bit value that is written to the specified destination register.

For information about using the assembler to program the various fields that the channel control registers, refer to "DMAMOV CCR" on page 16–46.

Operation

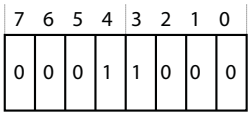
You can only use this instruction in a DMA channel thread.

DMANOP

No Operation does nothing. You can use this instruction for code alignment purposes.

Figure 16–16 shows the instruction encoding.

Figure 16–16. DMANOP Encoding



Assembler syntax

```
DMANOP
```

Operation

You can use the instruction with the DMA manager thread and the DMA channel thread.

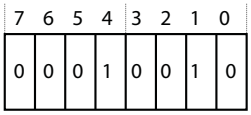
DMARMB

Read Memory Barrier forces the DMA channel to wait until all of the executed `DMALD` instructions for that channel have been issued on the AXI master interface and have completed.

This enables write-after-read sequences to the same address location with no hazards.

Figure 16–17 shows the instruction encoding.

Figure 16–17. DMARMB Encoding



Assembler syntax

```
DMARMB
```

Operation

You can only use this instruction in a DMA channel thread.

DMASEV

Send Event instructs the DMAC to modify an event-interrupt resource. Depending on how you program the interrupt enable register, this either:

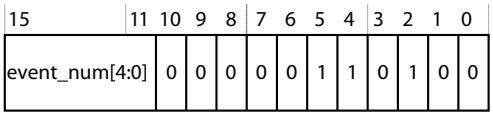
- Generates event <*event_num*>

  Typically, you use DMAWFE to stall a thread and then another thread executes DMASEV, using the appropriate event number, to unstall the waiting thread. Refer to "Using an Event to Restart DMA Channels" on page 16–15.

- Signals an interrupt using irq<*event_num*>.

Figure 16–18 shows the instruction encoding.

Figure 16–18. DMASEV Encoding



Assembler syntax

```
DMASEV <event_num>
```

where:

<*event_num*> 5-bit immediate, value 0-31

The DMAC aborts the thread if you select an event number that is not available.

Operation

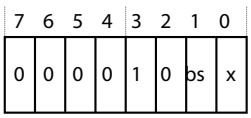
You can use the instruction with the DMA manager thread and the DMA channel thread. For more information, refer to "Using Events and Interrupts" on page 16–15.

DMAST[S | B]

Store instructs the DMAC to transfer data from the FIFO buffer to the location that the destination address registers specifies, using AXI transactions that the DA register and channel control registers specify. If the dst_inc bit in the channel control registers is set to incrementing, the DMAC updates the destination address registers after it executes DMAST[S|B].

Figure 16–19 shows the instruction encoding.

Figure 16–19. DMAST[S|B] Encoding



Assembler syntax

`DMAST[S|B]`

where:

`[S]` If S is present, the assembler sets `bs` to 0 and `x` to 1. The instruction is conditional on the state of the `request_type` flag:

- `request_type` = Single
  - The DMAC performs a `DMAST` instruction and it sets `awlen[3:0]`=0x0 so that the AXI write transaction length is one. The DMAC ignores the v value of the dst_burst_len field in the channel control registers.
- `request_type` = Burst
  - The DMAC performs a `DMANOP` instruction. The DMAC increments the channel PC to the next instruction. No state change occurs.

`[B]` If B is present, the assembler sets `bs` to 1 and `x` to 1. The instruction is conditional on the state of the `request_type` flag:

- `request_type` = Single
  - The DMAC performs a `DMANOP` instruction. The DMAC increments the channel PC to the next instruction. No state change occurs.
- `request_type` = Burst
  - The DMAC performs a `DMAST`.
  - If you do not specify the S or B operand, the assembler sets `bs` to 0 and `x` to 0, and the DMAC always executes a DMA store.

The DMAC sets the value of the `request_type` flag when it executes a `DMAWFP` instruction. Refer to "DMAWFP" on page 16–43.

Operation

You can only use this instruction in a DMA channel thread. If you specify the S or B operand, execution of the instruction is conditional on the state of the `request_type` flag matching that of the instruction.

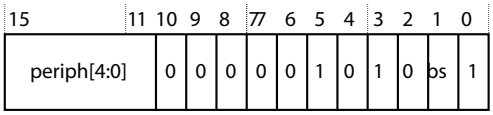
The DMAC only commences the burst when the MFIFO buffer contains all of the data necessary to complete the burst transfer.

DMASTP<S | B>

Store and notify Peripheral instructs the DMAC to transfer data from the FIFO buffer to the location that the destination address registers specifies, using AXI transactions that the DA register and channel control registers specify. It uses the DMA channel number to access the appropriate location in the FIFO buffer. After the DMA store is complete, and the DMAC has received a buffered write response, it issues an acknowledgement to the peripheral that the data transfer is complete. If the `dst_inc` bit in the channel control registers is set to incrementing, the DMAC updates the destination address registers after it executes `DMASTP<S|B>`.

Figure 16–20 shows the instruction encoding.

Figure 16–20. DMASTP<S|B> Encoding



Assembler syntax

```
DMASTP<S|B> <peripheral>
```

where:

<S> Sets bs to 0. This instructs the DMAC to perform:

- A single DMA store operation if request_type is programmed to Single

  The DMAC ignores the state of the dst_burst_len field in the channel control registers and always performs an AXI transfer with a burst length of one.

- A DMANOP if request_type is programmed to Burst.

<B> Sets bs to 1. This instructs the DMAC to perform:

- The DMA store if request_type is programmed to Burst
- A DMANOP if request_type is programmed to Single.

<peripheral> 5-bit immediate, value 0-31.

The DMAC sets the value of the request_type flag when it executes a DMAWFP instruction. Refer to "DMAWFP" on page 16–43.

Operation

You can only use this instruction in a DMA channel thread.

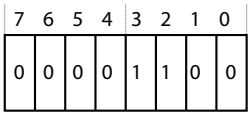
The DMAC only commences the burst when the MFIFO buffer contains all of the data necessary to complete the burst transfer.

DMASTZ

Store Zero instructs the DMAC to store zeros, using AXI transactions that the destination address registers and channel control registers specify. If the dst_inc bit in the channel control registers is set to incrementing, the DMAC updates the destination address registers after it executes DMASTZ.

Figure 16–21 shows the instruction encoding.

Figure 16–21. DMASTZ Encoding



Assembler syntax

```
DMASTZ
```

Operation

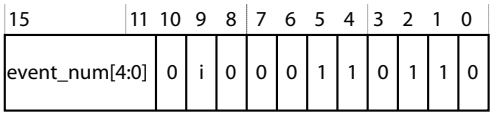
You can only use this instruction in a DMA channel thread.

DMAWFE

Wait For Event instructs the DMAC to halt execution of the thread until the event, that <*event_num*> specifies, occurs. When the event occurs, the thread moves to the Executing state and the DMAC clears the event. Refer to "Using Events and Interrupts" on page 16–15.

Figure 16–22 shows the instruction encoding.

Figure 16–22. DMAWFE Encoding



Assembler syntax

```
DMAWFE <event_num>[, invalid]
```

where:

<*event_num*> 5-bit immediate, value 0-31

`[invalid]` Sets `i` to 1. If `invalid` is present, the DMAC invalidates the instruction cache for the current DMA thread. If `invalid` is not present, then the assembler sets `i` to 0 and the DMAC does not invalidate the instruction cache for the current DMA.

The DMAC aborts the thread if you select an event number that is not available for your configuration of the DMAC.

To ensure cache coherency, you must use `invalid` when a processor writes the instruction stream for a DMA channel.

Operation

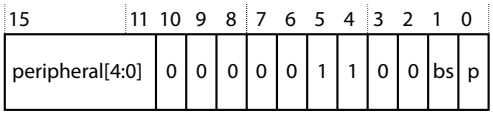
You can use the instruction with the DMA manager thread and the DMA channel thread.

DMAWFP

Wait For Peripheral instructs the DMAC to halt execution of the thread until the specified peripheral signals a DMA request for that DMA channel.

Figure 16–23 shows the instruction encoding.

Figure 16–23. DMAWFP Encoding



Assembler syntax

```
DMAWFP <peripheral>, <single|burst|periph>
```

where:

`<peripheral>` 5-bit immediate, value 0-31

The DMAC aborts the thread if you select a peripheral number that is not available.

`<single>` Sets `bs` to 0 and `p` to 0. This instructs the DMAC to continue executing the DMA channel thread after it receives a single or burst DMA request. The DMAC sets the `request_type` to Single, for that DMA channel.

`<burst>` Sets `bs` to 1 and `p` to 0. This instructs the DMAC to continue executing the DMA channel thread after it receives a burst DMA request. The DMAC sets the `request_type` to Burst.

The DMAC ignores single burst DMA requests.

`<periph>` Sets `bs` to 0 and `p` to 1. This instructs the DMAC to continue executing the DMA channel thread after it receives a single or burst DMA request. The DMAC sets the `request_type` to:

Single When it receives a single DMA request.

Burst When it receives a burst DMA request.

Operation

You can only use this instruction in a DMA channel thread.

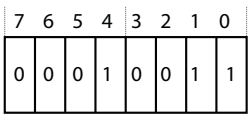
DMAWMB

Write Memory Barrier forces the DMA channel to wait until all of the executed `DMAST` instructions for that channel have been issued on the AXI master interface and have completed.

This permits read-after-write sequences to the same address location with no hazards.

Figure 16–24 shows the instruction encoding.

Figure 16–24. DMAWMB Encoding



Assembler syntax

```
DMAWMB
```

Operation

You can only use this instruction in a DMA channel thread.

Assembler Directives

The assembler provides the following additional commands:

- DCD
- DCB
- DMALP
- DMALPFE
- DMAMOV CCR

DCD

Assembler directive to place a 32-bit immediate in the instruction stream.

Syntax

```
DCD imm32
```

DCB

Assembler directive to place an 8-bit immediate in the instruction stream.

Syntax

```
DCB imm8
```

DMALP

Assembler directive to insert an iterative loop.

Syntax

```
DMALP [<LC0>|<LC1>] <loop_iterations>
```

where:

`<loop_iterations>`

An 8-bit value that specifies the number of loops to perform.

For clarity in writing assembler instructions, the 8-bit value is the actual number of iterations of the loop to be executed. The assembler decrements this by one to create the actual value, 0-255, that the DMAC uses.

`[LC0]` If `LC0` is present, the DMAC stores `<loop_iterations>` in the loop counter 0 registers.

`[LC1]` If `LC1` is present, the DMAC stores `<loop_iterations>` in the loop counter 1 registers.

If LC0 or LC1 is not present, the assembler determines the loop counter register to use.

DMALPFE

Assembler directive to insert a repetitive loop.

Syntax

```
DMALPFE
```

Enables the assembler to clear the nf bit that is present in DMALPEND. Refer to "DMALPEND[S | B]" on page 16–36.

DMAMOV CCR

Assembler directive that enables you to program the channel control registers using the specified format.

Syntax

```
DMAMOV CCR,
   [SB<1-16>] [SS<8|16|32|64|128>] [SA<I|F>]
   [SP<imm3>] [SC<imm4>]
   [DB<1-16>] [DS<8|16|32|64|128>] [DA<I|F>]
   [DP<imm3>] [DC<imm4>]
   [ES<8|16|32|64|128>]
```

Table 16–3 shows the argument descriptions and the default values.

Table 16–3. DMAMOV CCR argument description and the default values

Syntax	Description	Options	Default
SA	Source address increment. Sets the value of arburst[0]	I = Increment F = Fixed	I
SS	Source burst size in bits. Sets the value of arsize[2:0]	8, 16, 32, or 64	8
SB	Source burst length. Sets the value of arlen[3:0]	1 to 16	1
SP	Source protection	0 to 7 (1)	0
SC	Source cache	0 to 15 (1) (2)	0
DA	Destination address increment. Sets the value of awburst[0]	I = Increment F = Fixed	I
DS	Destination burst size in bits. Sets the value of awsize[2:0]	8, 16, 32, or 64	8
DB	Destination burst length. Sets the value of awlen[3:0]	1 to 16	1
DP	Destination protection	0 to 7 (1)	0
DC	Destination cache	0 to 15 (1) (3)	0
ES	Endian swap size, in bits	8, 16, 32, or 64	8

Notes to Table 16–3:

(1) You must use decimal values when programming this immediate value

(2) Because the DMAC ties ARCACHE[3] LOW, the assembler always sets bit 3 to 0 and uses bits [2:0] of your chosen value for SC.

(3) Because the DMAC ties AWCACHE[2] LOW, the assembler always sets bit 2 to 0 and uses bit [3] and bits [1:0] of your chosen value for DC.

MFIFO Buffer Usage Overview

This section shows MFIFO buffer usage for some example DMA channel programs.

About MFIFO Buffer Usage Overview

The MFIFO buffer is a shared resource that is utilized on a first-come, first-served basis by all currently active channels. To a program, it appears as a set of variable-depth parallel FIFO buffers, one per channel, with the restriction that the total depth of all the Fifes cannot exceed the buffer depth, 512. The width of the AXI master interface is the same as the MFIFO buffer width.

The DMAC is capable of realigning data from the source to the destination. For example, the DMAC shifts the data by two byte lanes when it reads a word from address 0x103 and writes to address 0x205. All byte manipulations occur when data enters the MFIFO buffer, as a result of an AXI read due to a `DMALD` instruction, so that the DMAC does not need to manipulate the data when it removes it from the MFIFO buffer, as a result of an AXI write due to a `DMAST` instruction. Therefore the storage and packing of the data in the MFIFO buffer is determined by the destination address and transfer characteristics.

When a program specifies that incrementing transactions are to be performed to the destination, the DMAC packs data into the MFIFO buffer to minimize the usage of the MFIFO buffer entries. For example, the DMAC packs two 32-bit words into a single entry in the MFIFO buffer when the DMAC has a 64-bit AXI data bus and the program uses a source address of 0x100, and destination address of 0x200.

In certain situations, the number of entries required to store the data loaded from a source is not a simple calculation of amount of source data divided by MFIFO buffer width. The calculation of the number of entries required is not simple when any of the following occur:

- The source address is not aligned to the AXI bus width.
- The destination address is not aligned to the AXI bus width.
- The transactions are to a fixed destination, that is, a non-incrementing address.

The `DMALD` and `DMAST` instructions each specify that an AXI transaction is to be performed. The amount of data transferred by an AXI transaction depends on the values programmed in to the `CCR`*n* register and the address of the transaction.

For information about unaligned transfers, refer to the *AMBA AXI Protocol Specification v1.0*, available from the ARM website (infocenter.arm.com).

The following sections provide several example DMAC programs together with illustrations of the MFIFO buffer usage.

These sections show MFIFO buffer usage in the following ways:

- a graph of the number of MFIFO buffer entries versus time
- a diagram of the byte-lane manipulation that the DMAC performs when data enters the MFIFO buffer.

The numbers 0 and 7 in the MFIFO buffer diagrams indicate the byte lanes in the MFIFO buffer.

Aligned Transfers

The following sections show examples of aligned transfers.

Simple Aligned Program

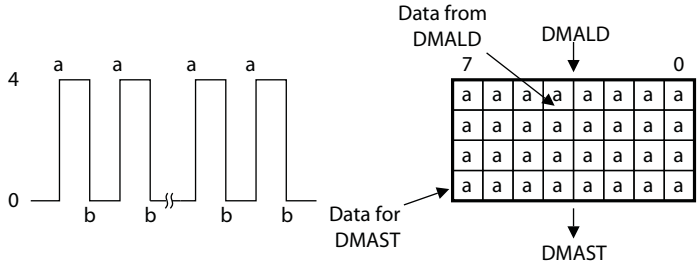
In this program, the source address and destination address are aligned with the AXI data bus width.

Example 16–2. Simple Aligned Program

```
DMAMOV CCR, SB4 SS64 DB4 DS64
DMAMOV SAR, 0x1000
DMAMOV DAR, 0x4000
DMALP 16
DMALD ; shown as a in Figure 16-25
DMAST ; shown as b in Figure 16-25
DMALPEND
DMAEND
```

Figure 16–25 shows the MFIFO buffer usage for this program.

Figure 16–25. Simple Aligned Program



In Figure 16–25, each DMALD requires four entries and each DMAST removes four entries.

This example has a static requirement of zero MFIFO buffer entries and a dynamic requirement of four MFIFO buffer entries.

Aligned Asymmetric Program with Multiple Loads

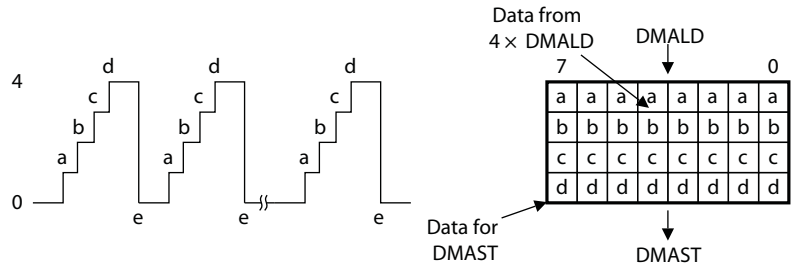
The following program performs four loads for each store and the source address and destination address are aligned with the AXI data bus width.

Example 16–3. Aligned Asymmetric Program with Multiple Loads

```
DMAMOV CCR, SB1 SS64 DB4 DS64
DMAMOV SAR, 0x1000
DMAMOV DAR, 0x4000
DMALP 16
DMALD ; shown as a in Figure 16-26
DMALD ; shown as b in Figure 16-26
DMALD ; shown as c in Figure 16-26
DMALD ; shown as d in Figure 16-26
DMAST ; shown as e in Figure 16-26
DMALPEND
DMAEND
```

Figure 16–26 shows the MFIFO buffer usage for this program.

Figure 16–26. Aligned Asymmetric Program with Multiple Loads



In Figure 16–26, each DMALD requires one entry and each DMAST removes four entries.

This example has a static requirement of zero MFIFO buffer entries and a dynamic requirement of four MFIFO buffer entries.

Aligned Asymmetric Program with Multiple Stores

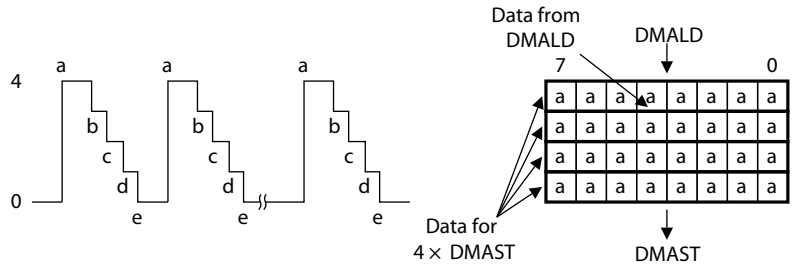
The following program performs four stores for each load and the source address and destination address are aligned with the AXI data bus width.

Example 16–4. Aligned Asymmetric Program with Multiple Stores

```
DMAMOV CCR, SB4 SS64 DB1 DS64
DMAMOV SAR, 0x1000
DMAMOV DAR, 0x4000
DMALP 16
DMALD ; shown as a in Figure 16-27
DMAST ; shown as b in Figure 16-27
DMAST ; shown as c in Figure 16-27
DMAST ; shown as d in Figure 16-27
DMAST ; shown as e in Figure 16-27
DMALPEND
DMAEND
```

Figure 16–27 shows the MFIFO buffer usage for this program.

Figure 16–27. Aligned Asymmetric Program with Multiple Stores



In Figure 16–27, each DMALD requires four entries and each DMAST removes one entry.

This example has a static requirement of zero MFIFO buffer entries and a dynamic requirement of four MFIFO buffer entries.

Unaligned Transfers

The following sections show examples of unaligned transfers.

Aligned Source Address to Unaligned Destination Address

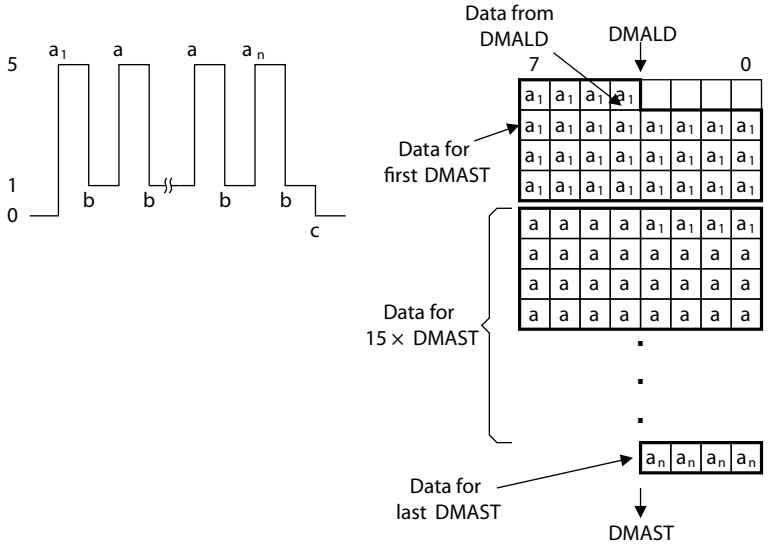
In this program, the source address is aligned with the AXI data bus width but the destination address is unaligned. The destination address is not aligned to the destination burst size so the first DMAST instruction removes less data than the first DMALD instruction reads. Therefore, a final DMAST of a single word is required to clear the data from the MFIFO buffer.

Example 16–5. Aligned Source Address to Unaligned Destination Address

```
DMAMOV CCR, SB4 SS64 DB4 DS64
DMAMOV SAR, 0x1000
DMAMOV DAR, 0x4004
DMALP 16
DMALD ; shown as a1, ... a, an in Figure 16-28
DMAST ; shown as b in Figure 16-28
DMALPEND
DMAMOV CCR, SB4 SS64 DB1 DS32
DMAST ; shown as c in Figure 16-28
DMAEND
```

Figure 16–28 shows the MFIFO buffer usage for this program.

Figure 16–28. Aligned to Unaligned Program



The first DMALD instruction loads four doublewords but because the destination address is unaligned, the DMAC shifts them by four bytes and therefore it uses five entries in the MFIFO buffer.

Each DMAST requires only four entries of data and therefore the extra entry remains in use for the duration of the program until it is emptied by the last DMAST.

This example has a static requirement of one MFIFO buffer entry and a dynamic requirement of four MFIFO buffer entries.

Unaligned Source Address to Aligned Destination Address

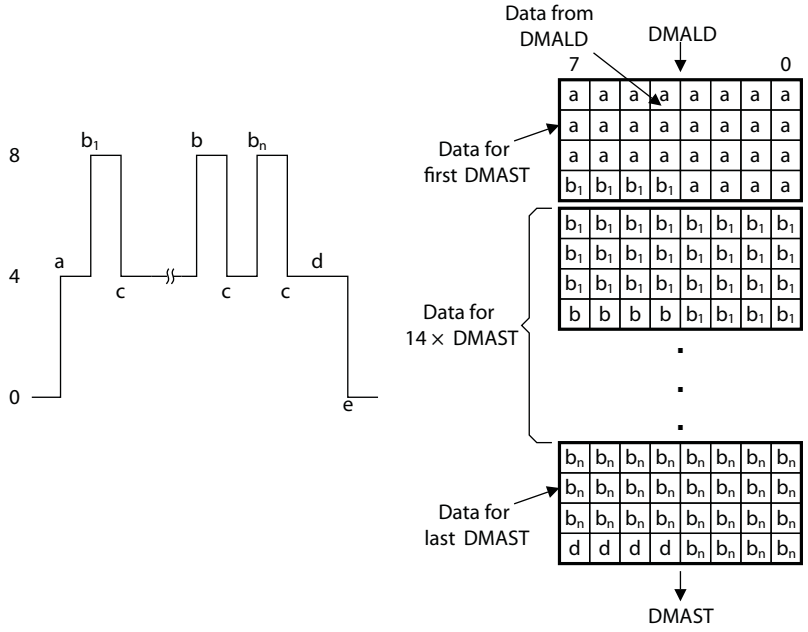
In this program, the source address is unaligned with the AXI data bus width but the destination address is aligned. The source address is not aligned to the source burst size so the first DMALD instruction reads in less data than the DMAST. Therefore, an extra DMALD is required to satisfy the first DMAST.

Example 16–6. Unaligned Source Address to Aligned Destination Address

```
DMAMOV CCR, SB4 SS64 DB4 DS64
DMAMOV SAR, 0x1004
DMAMOV DAR, 0x4000
DMALD ; shown as a in Figure 16-29
DMALP 15
DMALD ; shown as b1, ... b, bn in Figure 16-29
DMAST ; shown as c in Figure 16-29
DMALPEND
DMAMOV CCR, SB1 SS32 DB4 DS64
DMALD ; shown as d in Figure 16-29
DMAST ; shown as e in Figure 16-29
DMAEND
```

Figure 16–29 shows the MFIFO buffer usage for this program.

Figure 16–29. Unaligned to Aligned Program



The DMALD shown as **d** does not increase the MFIFO buffer usage because it loads four bytes into an MFIFO buffer entry that the DMAC has already allocated to this channel.

The first DMALD instruction does not load sufficient data to enable the DMAC to execute a DMAST and therefore the program includes an additional DMALD, prior to the start of the loop. After the first DMALD, the subsequent DMALDs align with the source burst size. This optimizes the p performance but it requires a larger number of MFIFO buffer entries.

This example has a static requirement of four MFIFO buffer entries and a dynamic requirement of four MFIFO buffer entries.

Unaligned Source Address to Aligned Destination Address with Excess Initial Load

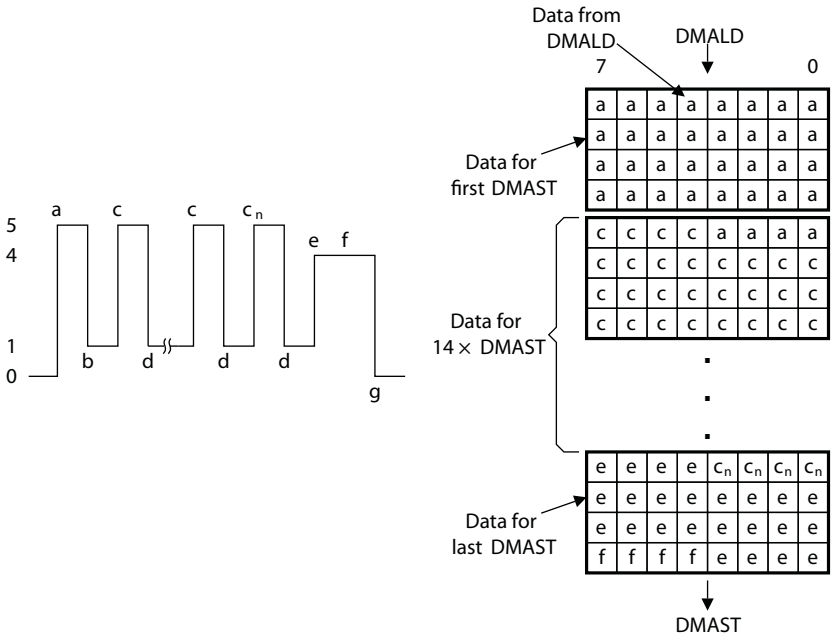
This program is an alternative to that described in "Unaligned Source Address to Aligned Destination Address" on page 16–51. The program uses a different sequence of source bursts which might be less efficient but requires fewer MFIFO buffer entries.

Example 16–7. Unaligned Source Address to Aligned Destination Address with Excess Initial Load

```
DMAMOV CCR, SB5 SS64 DB4 DS64
DMAMOV SAR, 0x1004
DMAMOV DAR, 0x4000
DMALD ; shown as a in Figure 16-30
DMAST ; shown as b in Figure 16-30
DMAMOV CCR, SB4 SS64 DB4 DS64
DMALP 14
DMALD ; shown as c and cn in Figure 16-30
DMAST ; shown as d in Figure 16-30
DMALPEND
DMAMOV CCR, SB3 SS64 DB4 DS64
DMALD ; shown as e in Figure 16-30
DMAMOV CCR, SB1 SS32 DB4 DS64
DMALD ; shown as f in Figure 16-30
DMAST ; shown as g in Figure 16-30
DMAEND
```

Figure 16–30 shows the MFIFO buffer usage for this program.

Figure 16–30. Unaligned to Aligned with Excess Initial Load



The DMALD shown as **f** does not increase the MFIFO buffer usage because it loads four bytes into an MFIFO buffer entry that the DMAC has already allocated to this channel.

The first DMALD instruction loads five beats of data to enable the DMAC to execute the first DMAST.

After the first DMALD, the subsequent DMALDs are not aligned to the source burst size, for example the second DMALD reads from address 0x1028. After the loop, the final two DMALDs read the data required to satisfy the final DMAST.

This example has a static requirement of one MFIFO buffer entry and a dynamic requirement of four MFIFO buffer entries.

Aligned Burst Size Unaligned MFIFO Buffer

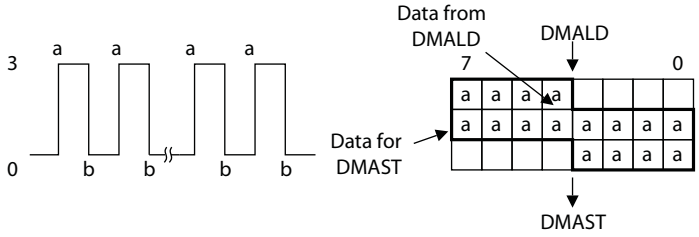
In this program, the destination address, which is narrower than the MFIFO buffer width, aligns with the burst size but does not align with the MFIFO buffer width.

Example 16–8. Aligned Burst Size Unaligned MFIFO Buffer

```
DMAMOV CCR, SB4 SS32 DB4 DS32
DMAMOV SAR, 0x1000
DMAMOV DAR, 0x4004
DMALP 16
DMALD ; shown as a in Figure 16-31
DMAST ; shown as b in Figure 16-31
DMALPEND
DMAEND
```

Figure 16–31 shows the MFIFO buffer usage for this program.

Figure 16–31. Aligned Burst with Unaligned MFIFO Buffer Width



In this example, the destination address is not 64-bit aligned, it requires three rather than the expected two MFIFO buffer entries.

This example has a static requirement of zero MFIFO buffer entries and a dynamic requirement of three MFIFO buffer entries.

Fixed Transfers

The following section shows an example of a fixed destination with aligned address.

Fixed Destination with Aligned Address

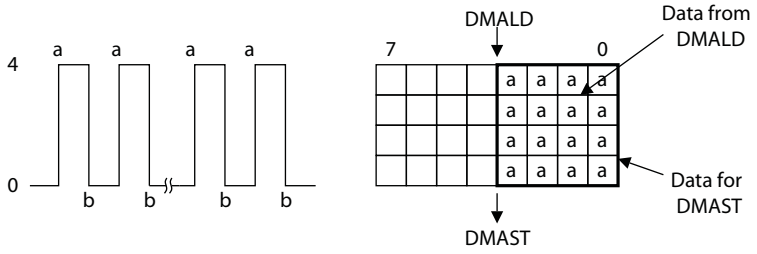
In this program, the source address and destination address are aligned with the AXI data bus width, and the destination address is fixed.

Example 16–9. Fixed Destination with Aligned Address

```
DMAMOV CCR, SB2 SS64 DB4 DS32 DAF
DMAMOV SAR, 0x1000
DMAMOV DAR, 0x4000
DMALP 16
DMALD ; shown as a in Figure 16-32
DMAST ; shown as b in Figure 16-32
DMALPEND
DMAEND
```

Figure 16–32 shows the MFIFO buffer usage for this program.

Figure 16–32. Fixed Destination with Aligned Address



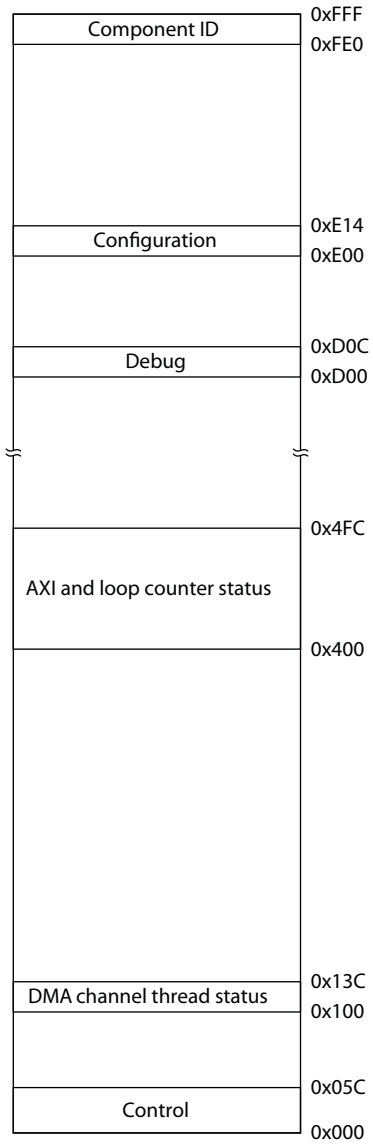
Each DMALD in the program loads two 64-bit data transfers into the MFIFO buffer. Because the destination address is a 32-bit fixed address then the DMAC splits each 64-bit data item across two entries in the MFIFO buffer.

This example has a static requirement of zero MFIFO buffer entries and a dynamic requirement of four MFIFO buffer entries.

DMA Controller Registers

The register map of the DMAC spans a 4 KB region as shown in Figure 16–33.

Figure 16–33. DMAC Summary Register Map



In Figure 16–33, the register map consists of the following sections.

- 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:
  - Allows you to send instructions to a thread when debugging the program code.
  - Allows system firmware to send instructions to the DMA manager thread, as "Issuing Instructions to the DMAC using a Slave Interface" on page 16–9 describes.
- Configuration registers—enable system firmware to discover 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 unpredictable behavior.

Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the following links for the module instance:

- **dmanonsecure**
- **dmasecure**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 16–4 shows the revision history for this document.

Table 16–4. Document Revision History

Date	Version	Changes
November 2012	1.1	Minor updates.
January 2012	1.0	Initial release.

17. Ethernet Media Access Controller

cv_54017-1.2

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).

For complete information about IEEE 802.3 MAC, refer to *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 website (standards.ieee.org/findstds/).

Features of the Ethernet MAC

Here is a summary of the features supported by the EMAC peripheral.

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
- 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 16 KB)
- Programmable inter-frame gap (IFG), from 40 to 96 bit times in steps of 8

PHY Interface

- Reduced Gigabit Media Independent Interface (RGMII) for 10/100/1000
- Management Data Input/Output (MDIO) (IEEE 802.3) or I^2C PHY management interface

DMA Interface

- 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

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)

Other Features

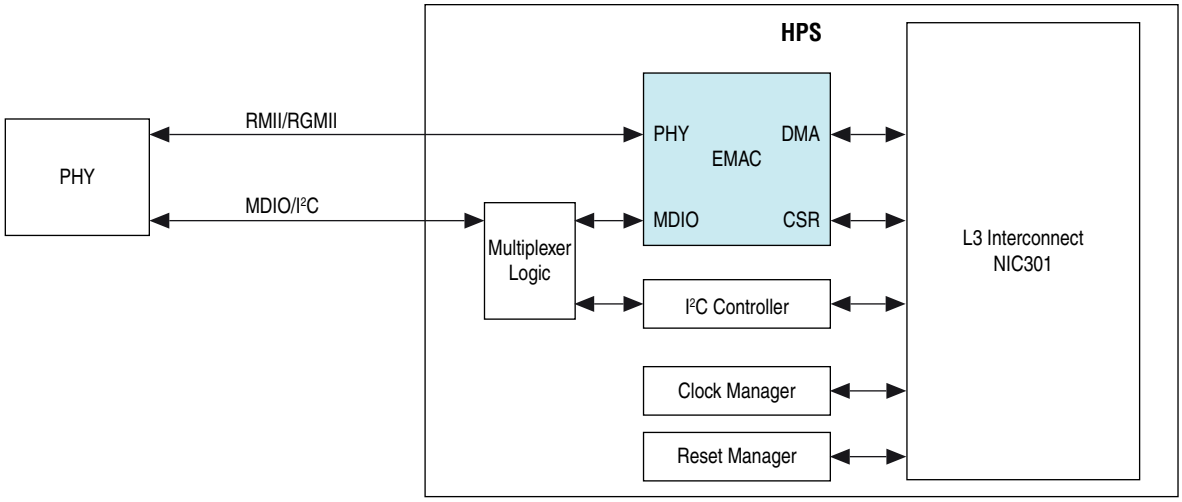
- Supports a variety of flexible address filtering modes
- Up to 31 additional 48-bit perfect destination address (DA) filters with masks for each byte
- Up to 31 48-bit source address (SA) comparison check with masks for each byte

- 256-bit hash filter (optional) for multicast and uni-cast 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

EMAC Block Diagram and System Integration

The EMACs are integrated into the HPS portion of the system on a chip (SoC) FPGA device. They communicate with the I/O pins. Figure 17–1 on page 17–3 shows the EMAC integration from a high level point of view.

Figure 17–1. EMAC System Integration



EMAC to RGMII Interface

The PHY datapath I/O is described in Table 17–1.

Table 17–1. External PHY Data Interface (Part 1 of 2)

EMAC Port	I/O	Width	Description
clk_tx_i	In	1	Transmit Clock. This is the transmit clock (125/25/2.5 MHz in 1G/100M/10Mbps) provided for the RGMII. All PHY transmit signals generated by the EMAC are synchronous to this clock.
phy_txd_o	Out	8	PHY Transmit Data.This is a group of eight transmit data signals driven by the MAC. Unused bits in the RGMII interface configuration are tied to low. RGMII: Bits [3:0] provide the RGMII transmit data. The data bus changes with both rising and falling edges of the transmit clock (clk_tx_i). The validity of the data is qualified with phy_txen_o. Synchronous to: clk_tx_i, clk_tx_180_i
phy_txen_o	Out	1	PHY Transmit Data Enable. This signal is driven by the EMAC component. RGMII: This signal is the control signal (rgmii_tctl) for the transmit data, and is driven on both edges of the clock. Synchronous to: clk_tx_i, clk_tx_180_i

Table 17–1. External PHY Data Interface (Part 2 of 2)

EMAC Port	I/O	Width	Description
rst_clk_tx_n_o	Out	1	Transmit clock reset output.
clk_rx_i	In	1	Receive clock. Clock frequency is 125/25/2.5 MHz in 1G/100M/10Mbps modes. It is provided by the external PHY. All PHY signals received by the EMAC are synchronous to this clock.
phy_rxd_i	In	8	PHY Receive Data.This is a bundle of eight data signals received from the PHY. RGMII: Bits [3:0] provide the RGMII receive data. The data bus is sampled with both rising and falling edges of the receive clock (clk_rx_i). The validity of the data is qualified with phy_rxdv_i. Synchronous to: clk_rx_i, clk_rx_180_i
phy_rxdv_i	In	1	PHY Receive Data Valid.This signal is driven by PHY. RGMII: This is the receive control signal used to qualify the data received on phy_rxd. This signal is sampled on both edges of the clock. Synchronous to: clk_rx_i, clk_rx_180_i
rst_clk_rx_n_o	Out	1	Receive clock reset output.
phy_intf_sel_i[1:0]	In	2	PHY Interface Select: These pins select one of the PHY interfaces of the EMAC. This is sampled only during reset assertion and ignored after that. ■ 01: RGMII ■ 00, 10, and 11: Invalid
clk_ref_i	In	1	This is the reference clock to the EMAC. The clock is emac0_clk or emac1_clk supplied by the clock manager. The system manager drives the phy_intf_sel signal to control which clock is used. The clock rate is 250 MHz.

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.

Table 17–2. PHY MDIO Management Interface

Signal	I/O	Width	Description
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 gmii_mdc_o clock.
gmii_mdo_o	Out	1	Management Data Out.The EMAC uses this signal to transfer control and data information to the PHY.

Table 17–2. PHY MDIO Management Interface

Signal	I/O	Width	Description
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.
gmii_mdc_o	Out	1	Management Data Clock. The EMAC provides timing reference for the gmii_mdi_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 can use two of the four general purpose I²C peripherals for controlling the PHY devices.

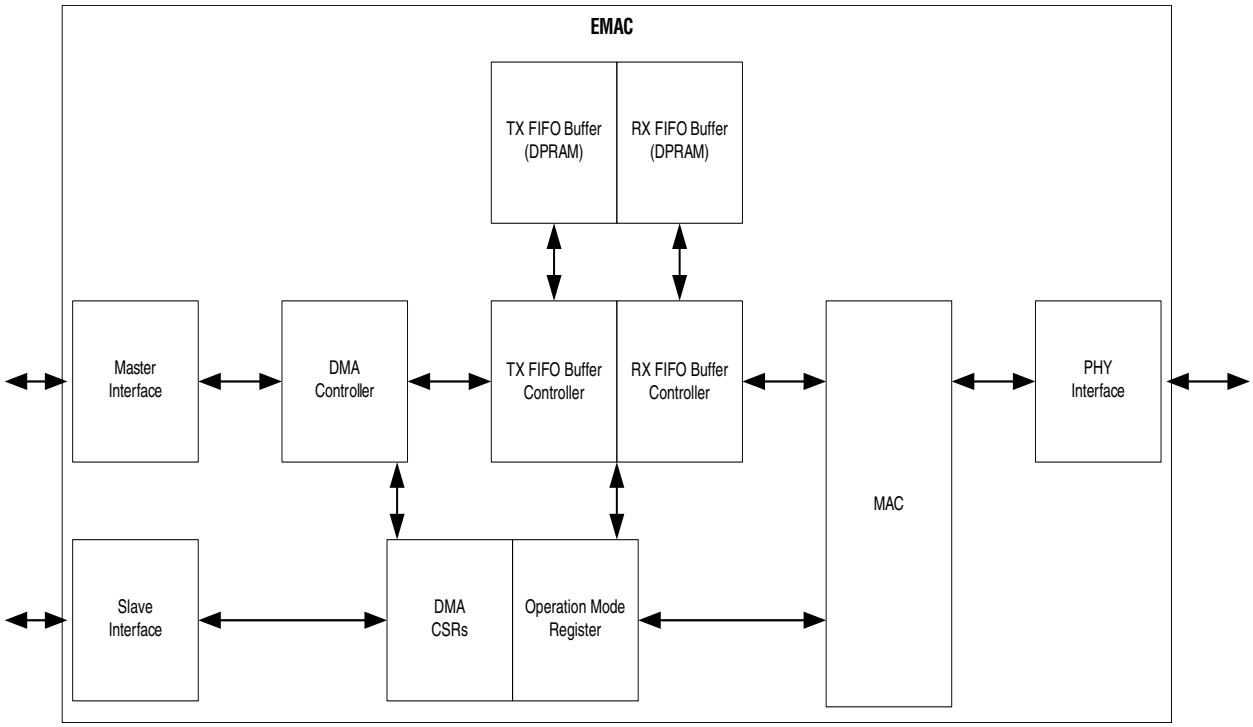
IEEE 1588

The EMAC supports IEEE 1588 operation in all modes with a resolution of one µs. It can be used by the ARM® Cortex™-A9 microprocessor unit (MPU) subsystem to maintain synchronization between the time counters internal to the two MACs.

Functional Description of the EMAC

Figure 17–2 illustrates a high level block diagram of the EMAC with its interfaces.

Figure 17–2. EMAC Block Diagram



There are two host interfaces to the MAC. The management host interface, a 32-bit slave interface, provides access to the CSR set. The data interface is a 32-bit interface. It controls data transfer between the direct memory access (DMA) controller channels and the rest of the HPS system through the NIC-301 L3 interconnect.

There is a built-in DMA controller which 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 controller. On transmit, the Ethernet frames read into the transmit FIFO buffer (1024 x 42 bits), and eventually trigger the MAC to perform the transfer. Received Ethernet frames are stored in the receive FIFO buffer, also indicating the FIFO buffer fill level to the DMA controller. The DMA controller then initiates the configured burst transfers. Both receive and transmit transfer status are taken from the MAC and transferred to the DMA.

Host Interfaces

There are two host interfaces in the EMAC: a slave and a master. The master is connected to the L3 master peripheral switch interface in the L3 interconnect block.

Slave

The EMAC CSR set access is provided by a slave interface. The slave is connected to the level 4 (L4) bus.

Master

The DMA interface is provided by a master interface. 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. The only exceptions to this are 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.

Write data transfers are generally performed as posted writes with OK responses returned as soon as the 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 which generated the original request and asserts an interrupt signal. The host needs to 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. This is important when resource contention bottlenecks arise under high system load conditions.

Cache Control Interface

The system manager provides the values for the master cache outputs through this interface. These inputs are used as the outputs to the L3 interconnect extending the capabilities of this block with respect to the cacheable characteristics of master transfers.

To configure EMAC DMA controller to perform cacheable accesses, configure the cache bits in the system manager. Cache bits should only be accessed at boot time, before the EMAC controller is brought out of reset.

For more information, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

External PHY

The following PHY interfaces are supported for the HPS:

- RGMII for 10/100/1000

The EMAC also has a control interface used for configuration and status monitoring of the PHY. In this case, the PHY is the slave device. There are two choices of control interface:

- MDIO
- I^2C interface

The MDIO interface is built into the EMAC while the I^2C interface uses separate I^2C peripherals residing on the HPS. The interfaces are multiplexed externally to the EMAC.

Transmit and Receive Data FIFO Buffers

Each EMAC component has associated transmit and receive data FIFO buffer instances. Both FIFO buffer instances are 1024 x 42 bits. The FIFO buffer word consists of:

- Data: 32 bits
- Sideband:
  - End of frame (EOF): one bit
  - Byte enables (BE): two bits
  - Error correction code (ECC): seven bits

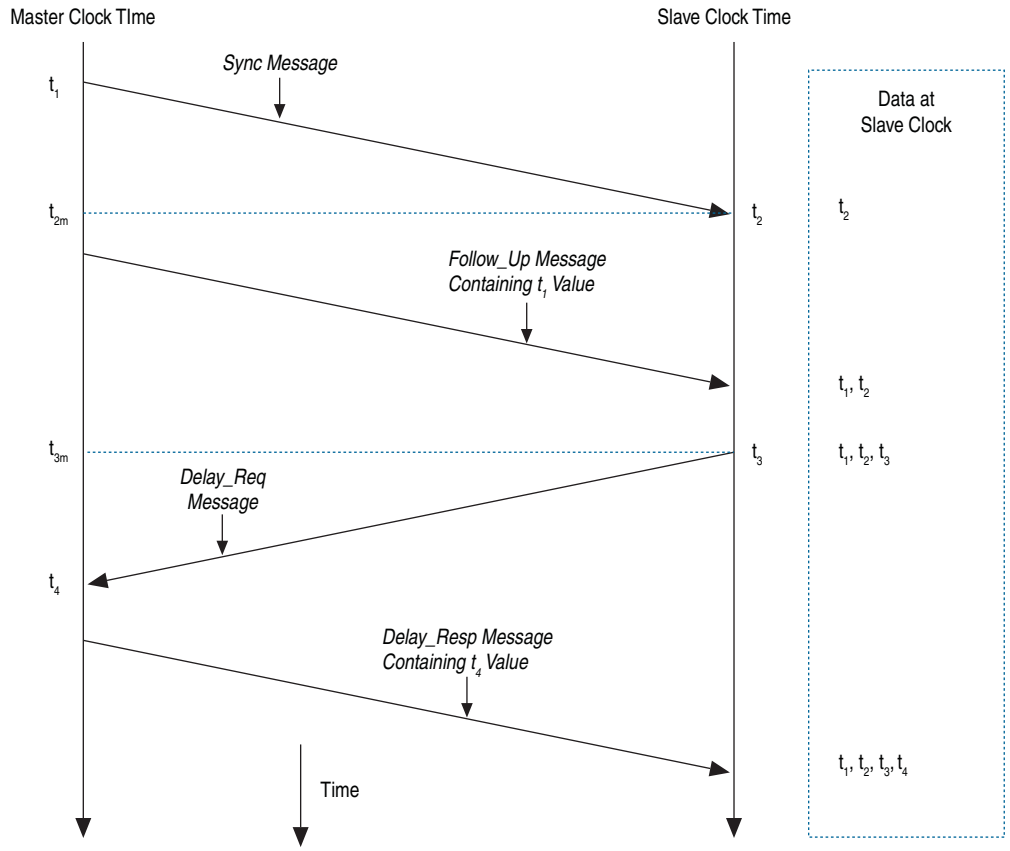
The data and sideband are protected by the seven-bit single error correct, double error detect (SEC-DED) code word. These FIFO buffer RAMs also contain ECC enable, error injection and status pins. The enable and error injection pins are inputs driven by the system manager and the status pins are outputs driven to the MPU subsystem.

IEEE 1588-2002 Time Stamps

The IEEE 1588-2002 standard defines the Precision Time Protocol (PTP) which 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. Figure 17–3 shows the process that PTP uses for synchronizing a slave node to a master node by exchanging PTP messages. †

Figure 17–3. Networked Time Synchronization



As shown in Figure 17–3, the PTP uses the following process: †

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 t1. 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, t2, using its timing reference. †
3. The master sends a Follow_up message to the slave, which contains t1 information for later use. †
4. The slave sends a Delay_Req message to the master, noting the exact time, t3, at which this frame leaves the PHY interface. †
5. The master receives the message, capturing the exact time, t4, at which it enters its system. †
6. The master sends the t4 information to the slave in the Delay_Resp message. †

7. The slave uses the four values of t1, t2, t3, and t4 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 one µs. 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 an alternate 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 updates the transmit descriptor accordingly. †

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 (standards.ieee.org). †

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, `osc1_clk`. The `osc1_clk` clock comes from the clock manager and the `emac_ptp_ref_clk` clock comes from the FPGA fabric. This time 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.

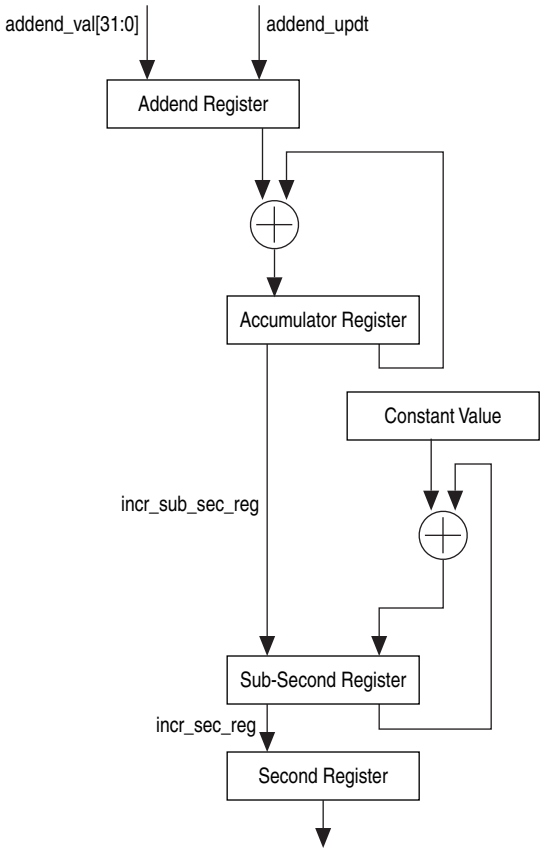
In the fine correction method, a slave clock's frequency drift with respect to the master clock is corrected over a period of time instead of in one clock, as in coarse correction. This helps maintain linear time and does not introduce drastic changes (or a large jitter) in the reference time between PTP sync message intervals. †

In this method, an accumulator sums up the contents of the `Timestamp_Addend` register, as shown in Figure 17–4. 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.

You must connect a PTP clock with a frequency higher than the frequency required for the specified accuracy. †

This algorithm is depicted in Figure 17–4: †

Figure 17–4. 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 MHz / 50 MHz = 1.32. Hence, the default addend value to be set in the register is 232 / 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 232 / 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 (232 / 1.32) must be programmed. †

In Figure 17–4, 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 $FreqCompensationValue_0$ in the Addend register. This value is as follows: †

$$FreqCompensationValue_0 = 232 / FreqDivisionRatio \dagger$$

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 $MasterSyncTime_n$ the master sends the slave clock a Sync message. The slave receives this message when its local clock is $SlaveClockTime_n$ and computes $MasterClockTime_n$ as: †

  $$MasterClockTime_n = MasterSyncTime_n + MasterToSlaveDelay_n \dagger$$

- The master clock count for current Sync cycle, $MasterClockCount_n$ is given by: †

  $$MasterClockCount_n = MasterClockTime_n – MasterClockTime_{n-1}$$

  (assuming that MasterToSlaveDelay is the same for Sync cycles n and n – 1) †

- The slave clock count for current Sync cycle, $SlaveClockCount_n$ is given by: †

  $$SlaveClockCount_n = SlaveClockTime_n – SlaveClockTime_{n-1} \dagger$$

- The difference between master and slave clock counts for current Sync cycle, $ClockDiffCount_n$ is given by: †

  $$ClockDiffCount_n = MasterClockCount_n – SlaveClockCount_n \dagger$$

- The frequency-scaling factor for slave clock, $FreqScaleFactor_n$ is given by: †

  $$FreqScaleFactor_n = (MasterClockCount_n + ClockDiffCount_n) / SlaveClockCount_n \dagger$$

- The frequency compensation value for Addend register, $FreqCompensationValue_n$ is given by: †

  $$FreqCompensationValue_n = FreqScaleFactor_n \times FreqCompensationValue_{n-1} \dagger$$

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 SFD of a frame is sent on the PHY interface. The frames for which you want to capture timestamps are controllable 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. The MAC does not process the transmitted frames to identify the PTP frames. You need to specify the frames for which you want to capture timestamps. The MAC returns the timestamp, along with the Transmit status of the frame, to hardware implemented in the FPGA. You can use the control bits in the transmit descriptor. The MAC returns the timestamp to the software inside the corresponding transmit descriptor, thus connecting the timestamp automatically to the specific PTP frame. †

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 IEEE 1588 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 (`osc1_clk`) 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. It means that the captured timestamp has the reference timing source value that is given 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 because of the PHY interface clocks 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, that is, from MAC clock domain to the FPGA clock domain. Therefore, a minimum delay is required between two consecutive timestamp captures. This delay is four clock cycles of the PHY interface and three clock cycles of PTP clocks. If the delay between two timestamp captures is less than this delay, 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 upon the operating mode and operating speed of the MAC as shown in Table 17–3. †

Table 17–3. 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 * PTP) + (4 * MII) <= 168 * MII, that is, ~0.5 MHz ((168 – 4) * 40 ns ÷ 3 = 2180 ns period)
1000-Mbps half-duplex operation	24 GMII clocks (4 for a jam pattern sent just after SFD because of collision + 12 IFG + 8 preamble) (1)	3 * PTP + 4 * GMII <= 24 * GMII, that is, 18.75 MHz

Notes to Table 17–3:

(1) For details about jam patterns, refer to *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 website (standards.ieee.org/findstds/).

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 only event messages. †
- 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. †

For details about the IEEE 1588-2008 standard, refer to *IEEE Standard 1588-2008 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, available on the IEEE Standards Association website (standards.ieee.org). †

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 †

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 only through the alternate (enhanced) 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. This controls the type of messages for which timestamps are taken. †

The DMA returns the timestamp to the software inside the corresponding transmit or receive descriptor. The advanced timestamp feature is supported only with the 32-byte alternate (enhanced) 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`. 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 a register to know which input's timestamp is available for reading at the top of this FIFO buffer. The MAC stores these timestamps in a FIFO buffer. Only 64 bits of the timestamp are stored in the FIFO buffer. When a timestamp is stored, the MAC indicates this to the host with an interrupt. The value of the timestamp is read through a FIFO buffer register access. †

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 does not support half-duplex mode. †

For details about the IEEE 802.3az Energy Efficient Ethernet standard, refer to the IEEE 802.3 Ethernet Working Group website (www.ieee802.org/3/). †

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 to the PHY. †

LPI Timers

Two timers internal to the EMAC are associated with LPI mode: †

- LPI Link Status (LS) Timer †
- LPI TW Timer †

The LPI LS timer counts, in ms, the time expired since the link status is 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 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. Therefore, the software can program up to 1023 ms. †

The LPI TW timer counts, in µs, 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 MAC supports the LPI TW timer in units of µs. The LPI TW timer is 16 bits wide. Therefore, the software can program up to 65535 µs. †

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 are shown below: †

- 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"
- "Multicast Destination Address Filter"
- "Hash or Perfect Address Filter"
- "Broadcast Address Filter"
- "Unicast Source Address Filter"
- "Inverse Filtering Operation (Invert the filter-match result at final output)"

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 64-bit hash table. It uses the upper six 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 64-bit hash table. For hash filtering, it uses the upper six 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. †

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. †

Non-TCP 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. †

Clocks and Resets

The Ethernet MAC controller uses the clocks shown in Table 17–4.

Table 17–4. Clocks

Name	Nominal Frequency	Functional Usage	Notes
clk_ref_i	250 Mhz	Reference Clock to the EMAC	If supplied from clock interface, clock is emac0_clk or emac1_clk
clk_tx_i	125/25/2.5 Mhz	Autonegotiates speed down to 10/100Mbps	
clk_rx_i		PHY provides this reference to MAC	All PHY signals received by the MAC are synchronous to this clock

Clock Gating for EEE

For the RGMII PHY interface, you can gate the transmit clock for Energy Efficient Ethernet (EEE) applications. For more information, refer to "Programming Guidelines for Energy Efficient Ethernet" on page 17–63.

Resets

The Ethernet MAC controller uses the reset signals shown in Table 17–5.

Table 17–5. Resets

Name	Nominal Frequency	Functional Usage	Notes
rst_clk_tx_n_o		Transmit clock reset output	Used to reset external PHY transmit clock domain logic
rst_clk_rx_n_o		Receive clock reset output	Used to reset external PHY receive clock domain logic

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

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. The controller use descriptors 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, and other normal/error conditions.

The DMA and the Host driver communicate through two data structures: †

- Control and Status registers (CSR) †
- Descriptor lists and data buffers †

For information about Control and Status registers, refer to "Ethernet MAC Address Map and Register Definitions" on page 17–67. Descriptors are described in "Normal Descriptor" on page 17–36 and "Alternate or Enhanced Descriptors" on page 17–47.

You can select an alternative descriptor structure during RTL configuration. The control bits in this descriptor structure are reassigned so that the application can use a larger buffer size (8 KB). For a detailed bit map of this alternative descriptor structure, refer to "Alternate or Enhanced Descriptors" on page 17–47. All descriptions in "DMA Controller" on page 17–21 refer to the default descriptor structure, not this new alternative. If you are using the alternate descriptor structure, ignore the descriptor-specific mapping in "DMA Controller" on page 17–21 and refer to the alternate descriptor-specific bit maps.

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[24] and TDES1[24]). 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. †

The descriptor ring and chain structures are shown in Figure 17–5 and Figure 17–6. †

Figure 17–5. Descriptor Ring Structure

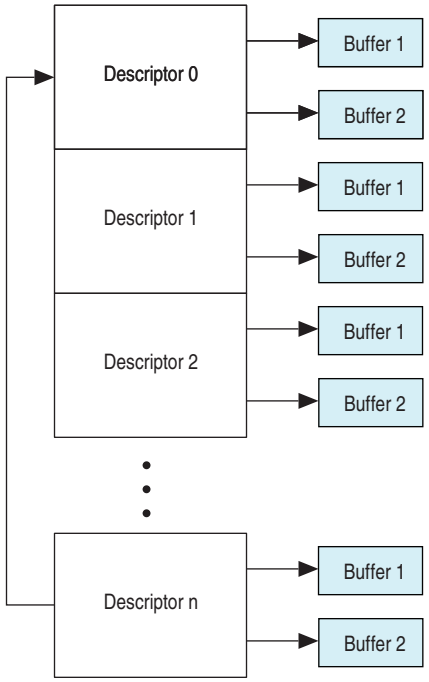
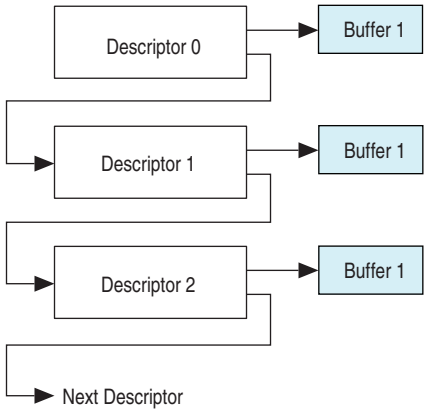


Figure 17–6. Descriptor Chain Structure



Initialization

Initialization for the EMAC is as follows.

1. Write to Register 0 (Bus Mode Register) to set Host bus access parameters. †
2. Write to Register 7 (Interrupt Enable Register) to mask unnecessary interrupt causes. †
3. Create the transmit and receive descriptor lists, and then write to DMA Register 3 (Receive Descriptor List Address Register) and Register 4 (Transmit Descriptor List Address Register), providing the DMA with the starting address of each list. †

4. Write to Register 1 (MAC Frame Filter), Register 2 (Hash Table High Register), and Register 3 (Hash Table Low Register) for desired filtering options. †
5. Write to Register 1 (MAC Frame Filter) to configure the operating mode and enable the transmit operation (Bit 3: Transmitter Enable). The PS and DM bits are set based on the auto-negotiation result (read from the PHY). †
6. Write to Register 6 (Operation Mode Register) to set Bits 13 and 1 to start transmission and reception. †
7. Write to Register 0 (MAC Configuration Register) to enable the receive operation (Bit 2: Receiver Enable). †

   The transmit and receive engines enter the Running state and attempt to acquire descriptors from the respective descriptor lists. The receive and transmit engines then begin processing receive and transmit operations. The transmit and receive processes are independent of each other and can be started or stopped separately. †

Host Bus Burst Access

The DMA attempts to execute fixed-length Burst transfers on the master interface if configured to do so through 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 PBL or 16 * 8/bus width) burst-size for the 16- bytes to be read.

The transmit DMA initiates a data transfer only when sufficient space to accommodate the configured burst is available in MTL transmit FIFO buffer or the number of bytes till the end of 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, then it transfers data using the best combination of INCR4, 8, or 16 and SINGLE transactions. Otherwise (no 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, then 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. Otherwise (FB bit of Register 0 (Bus Mode Register) is reset), it transfers data using INCR (undefined length) and SINGLE transactions.

When the interface is configured for address-aligned beats, both DMA engines ensure that the first burst transfer initiated is less than or equal to the size of the configured PBL. Thus, all subsequent beats start at an address that is aligned to the configured PBL. 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 (for 32-bit data bus), and 15 bytes need to 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 3 bytes of the last transfer are also ignored. The DMA always ensures it transfers a full 32-bit data to the MTL transmit FIFO buffer, unless it is the end-of-frame.

Example: Buffer Write

If the receive buffer address is 0x0000FF2 (for 64-bit data bus) and 16 bytes of a received frame need to be transferred, then the DMA writes 3 full words from address 0x00000FF0. But the first 2 bytes of first transfer and the last 6 bytes of the third 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 has to perform the size calculations. †

The transmit DMA transfers the exact number of bytes (indicated by buffer size field of TDES1) towards the MAC. If a descriptor is marked as 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 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 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 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. †

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

Transmission functions use transmit descriptors, described in detail in "Transmit Descriptor" on page 17–36.

TX DMA Operation: Default (Non-OSF) Mode

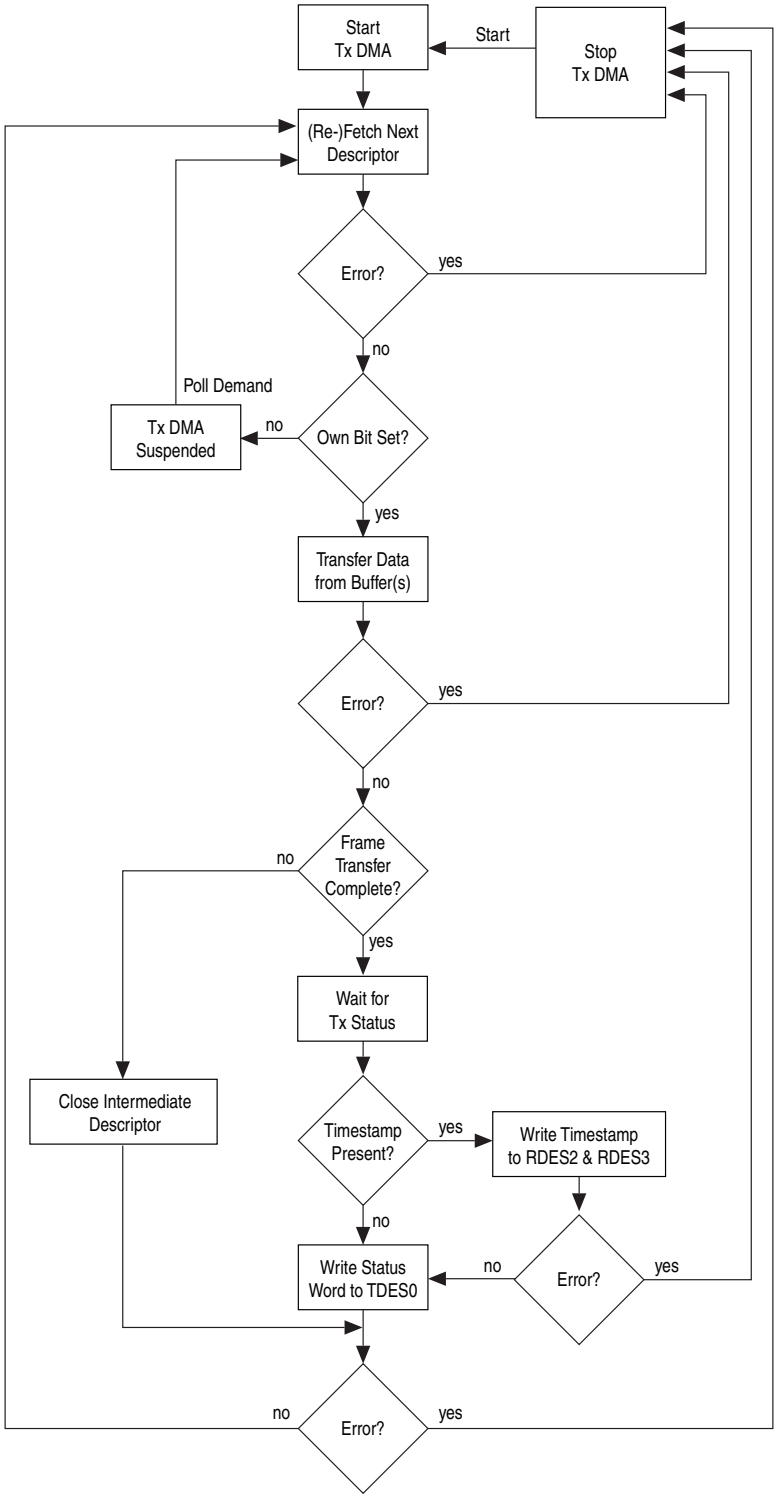
The transmit DMA engine in default mode proceeds as follows: †

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. Steps 3, 4, and 5 are repeated 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. †

The TX DMA transmission flow in default mode is shown in Figure 17–7. †

Figure 17–7. 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. †

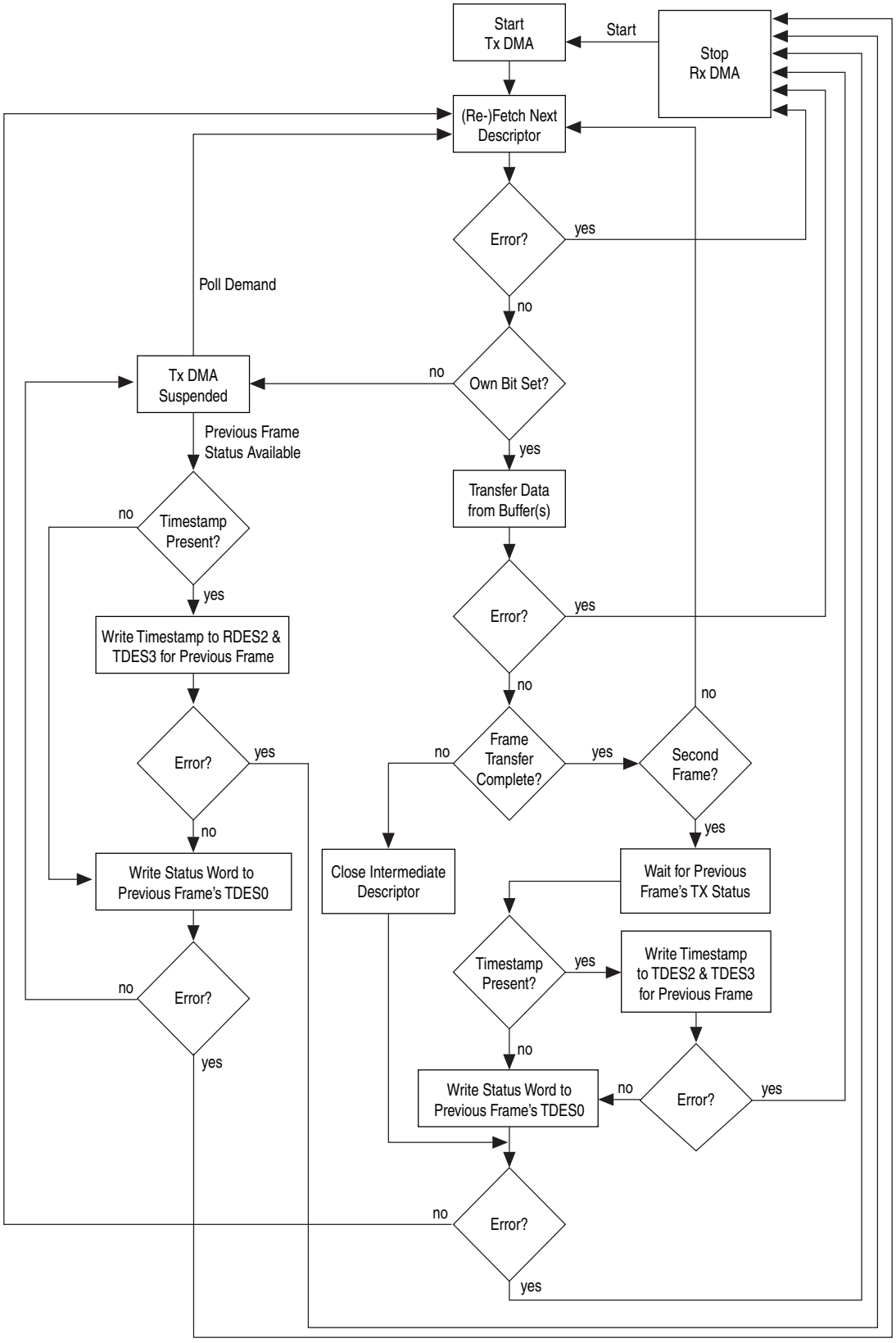
In OSF mode, the Run state transmit DMA operates in the following sequence: †

1. The DMA operates as described in steps 1–6 of "TX DMA Operation: Default (Non-OSF) Mode" on page 17–25. †
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)). †

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. †

The basic flow is described in Figure 17–8. †

Figure 17–8. 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. 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 data-chained 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 programmable 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. †

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

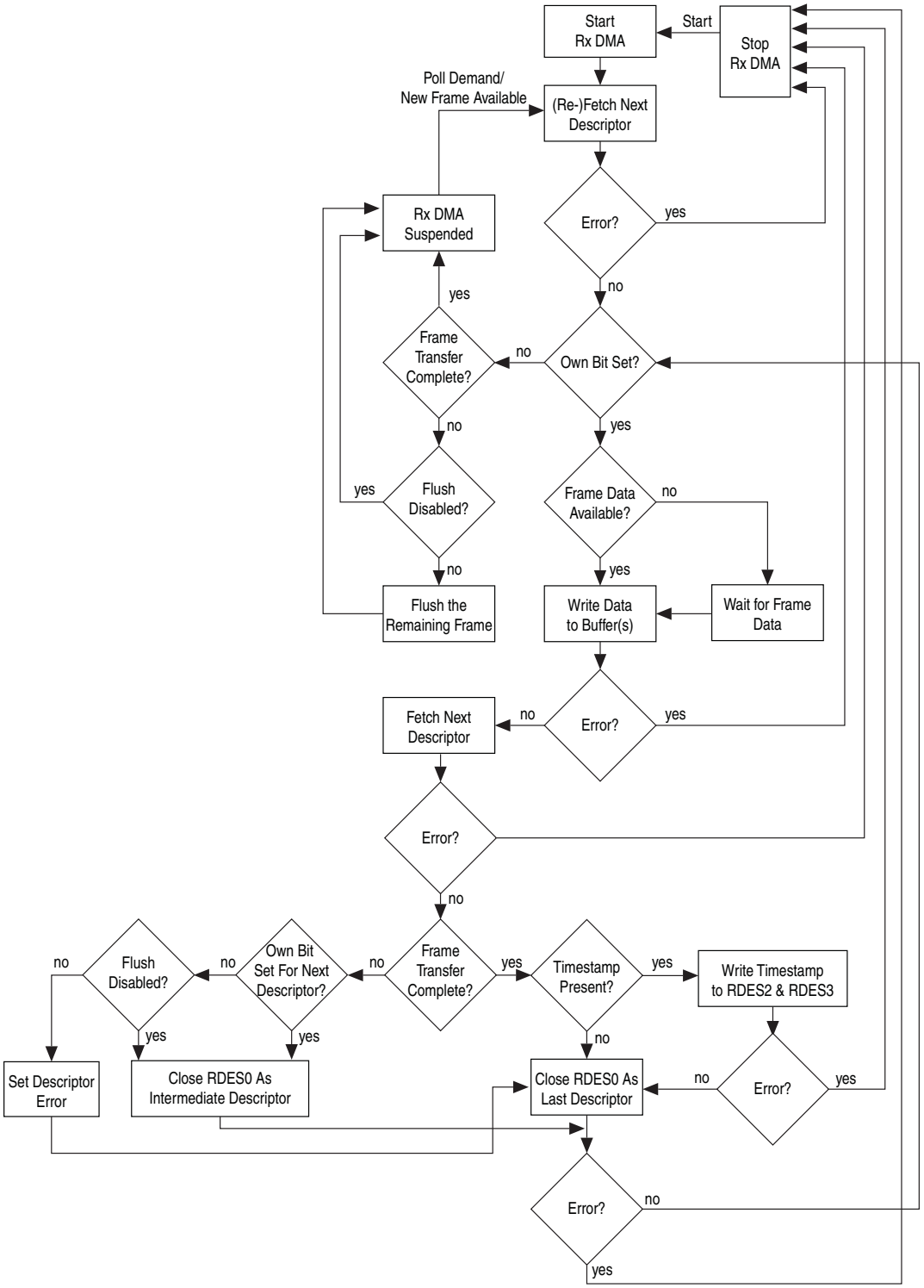
Receive functions use receive descriptors, described in detail in "Receive Descriptor" on page 17–40.

The receive DMA engine's reception sequence is depicted in Figure 17–9 on page 17–32 and 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 17–9. Receive DMA Operation



If software has enabled timestamping through CSR, when 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. Otherwise (that is, if timestamping is not enabled), the RDES2 and RDES3 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 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 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 First Descriptor (RDES0[9]) after the DMA Host interface becomes ready to receive a data transfer (if DMA is not fetching transmit data from the host), to delimit the frame. The descriptors are released when the Own (RDES[31]) bit is reset to 0, 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 (RDES[8]) and First Descriptor (RDES[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 the 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, the receive Process sets the Bit 7 (Receive Buffer Unavailable) of Register 5 (Status Register) and then 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 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. †

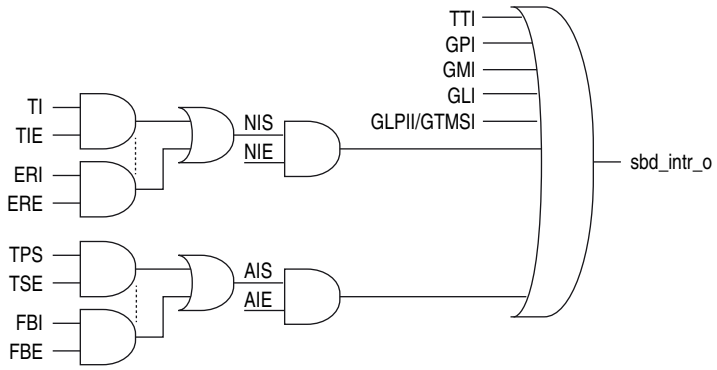
The discarding or flushing of the frame at the top of the MTL EX FIFO buffer can be avoided by disabling Flushing (Bit 24 of Register 6 (Operation Mode Register)). In such conditions, the receive process sets the Receive Buffer Unavailable status and returns to the Suspend state. †

Interrupts

Interrupts can be generated as a result of various events. The DMA Register 5 (Status Register) contains all the bits that might cause an interrupt. Register 7 (Interrupt Enable Register) contains an enable bit for each of the events that can cause an interrupt. †

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 de-asserted. If the MAC is the cause for assertion of the interrupt, then any of the GLI, GMI, GPI, TTI, or GLPII bits of Register 5 (Status Register) are set high, as shown in Figure 17–10.

Figure 17–10. sbd_intr_o Generation *(1)*



Note to Figure 17–10:

(1) Signals NIS and AIS are registered.

The 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 simultaneous, multiple events. The driver must scan the Register 5 (Status Register) for the cause of the interrupt. The interrupt is not generated again unless a new interrupting event occurs, after the driver has cleared the appropriate bit in Register 5 (Status Register). For example, the controller generates a Bit 6 (Receive Interrupt) of Register 5 (Status Register) and the driver begins reading Register 5 (Status Register). Next, Bit 7 (Receive Buffer Unavailable) of Register 5 (Status Register) occurs. The driver clears the receive interrupt. Even then, the `sbd_intr_o` signal is not de-asserted, because of the active or pending Receive Buffer Unavailable interrupt. †

Bits 7:0 (Interrupt Timer) of Register 9 (Receive Interrupt Watchdog Timer Register) is given for flexible control of 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] in Table 17–12 on page 17–43). When this timer runs out as per the programmed value, RI bit is set and the interrupt is asserted if the corresponding RI is enabled in Register 7 (Interrupt Enable Register). This timer gets disabled before it runs out, when a frame is transferred to memory and the RI is set because it is enabled for that descriptor. †

Error Response to DMA

For any data transfer initiated by a DMA channel, if the slave replies with an error response, that DMA stops all operations and updates the error bits and the Fatal Bus Error bit in the Register 5 (Status Register). The DMA controller can resume operation only after soft resetting or hard resetting the EMAC and reinitializing the DMA.

Descriptor Overview

This section describes the HPS EMAC DMA descriptors.

The DMA in the Ethernet subsystem transfers data based on a linked list of descriptors, as explained in "DMA Controller" on page 17–21. The descriptors are created in the system memory. The EMAC DMA supports the following two types of descriptors:

- Normal descriptor—This is the default descriptor format which can have 4 DWORDS. For information about this descriptor type, refer to "Normal Descriptor" on page 17–36. †
- Enhanced descriptor—This is the alternate descriptor format which can have 8 DWORDS (32 bytes). For information about this descriptor type, refer to "Alternate or Enhanced Descriptors" on page 17–47.

Each descriptor contains two buffers, two byte-count buffers, and two address pointers, which enable the adapter port to be compatible with various types of memory management schemes. Once configured, you cannot change the descriptor structure. †

Descriptor Endianness

The descriptor addresses must be aligned to the used bus width (Word, DWord, or LWord for 32-bit, 64-bit, or 128-bit bus respectively). The data bus is configured as little-endian.

The figures in this section show the normal transmit and receive descriptors. If you are using the enhanced descriptors, you must ensure that the descriptors are created accordingly in system memory. For more information about enhanced descriptors, refer to "Alternate or Enhanced Descriptors" on page 17–47. †

Normal Descriptor

This is the default descriptor format which can have 4 DWORDS. You can use this descriptor format when advanced features such as IEEE 1588 advanced timestamping are not enabled.

Transmit Descriptor

The DMA in the EMAC requires at least one descriptor for a transmit frame. In addition to two buffers, two byte-count buffers, and two address pointers, the transmit descriptor has control fields which can be used to control the MAC operation on per-transmit frame basis.

Transmit Descriptor 0 (TDES0)

TDES0 contains the transmitted frame status and the descriptor ownership information.

Table 17–6. Transmit Descriptor 0 (Part 1 of 3)

Bit	Description
31	OWN: Own Bit When set, this bit indicates that the descriptor is owned by the DMA. When this bit is reset, 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 empty. The ownership bit of the First descriptor of the frame should be set after all subsequent descriptors belonging to the same frame have been set. This avoids a possible race condition between fetching a descriptor and the driver setting an ownership bit. †
30:18	Reserved †
17	TTSS: TX Timestamp Status This status bit indicates that a timestamp has been captured for the corresponding transmit frame. When this bit is set, TDES2 and TDES3 have timestamp values that were captured for the transmit frame. This field is valid only when the Last Segment control bit (TDES1[30]) in a descriptor is set.
16	IHE: IP Header Error When set, this bit indicates that the Checksum Offload engine detected an IP header error and consequently did not modify the transmitted frame for any checksum insertion.

Table 17–6. Transmit Descriptor 0 (Part 2 of 3)

Bit	Description
15	ES: Error Summary Indicates the logical OR of the following bits: ■ TDES0[16]: IP Header Error ■ TDES0[14]: Jabber Timeout ■ TDES0[13]: Frame Flush ■ TDES0[12]: Payload Checksum Error ■ 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 †
14	JT: Jabber Timeout When set, this bit indicates that the MAC transmitter has experienced a jabber time-out. This bit is only set when the JD bit of Register 0 (MAC Configuration Register) is not set. †
13	FF: Frame Flushed When set, this bit indicates that the DMA or MAC transaction layer (MTL) flushed the frame because of a software flush command given by the CPU. †
12	12 PCE: Payload Checksum Error This bit, when set, indicates that the Checksum Offload engine had a failure and did not insert any checksum into the encapsulated TCP, UDP, or ICMP payload. This failure can be either because of insufficient bytes, as indicated by the IP Header's Payload Length field, or the MTL starting to forward the frame to the MAC transmitter in Store-and-Forward mode without the checksum having been calculated yet. This second error condition only occurs when the transmit FIFO buffer depth is less than the length of the Ethernet frame being transmitted to avoid deadlock, the MTL starts forwarding the frame when the FIFO buffer is full, even in the store-and-forward mode.
11	LC: Loss of Carrier When set, this bit indicates that 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 is valid only for the frames transmitted without collision and when the MAC operates in the half-duplex mode. †
10	NC: No Carrier When set, this bit indicates that the carrier sense signal form the PHY was not asserted during transmission. †
9	Reserved
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 was a VLAN-type frame. †
6:3	CC: Collision Count This 4-bit counter value indicates the number of collisions occurring before the frame was transmitted. The count is not valid when the Excessive Collisions bit (TDES0[8]) is set. †

Table 17–6. Transmit Descriptor 0 (Part 3 of 3)

Bit	Description
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 1000-Mbps mode, or in Jumbo frame enabled mode) if Bit 4 (Deferral Check) is set high in Register 0 (MAC Configuration Register). †
1	UF: Underflow Error When set, this bit indicates that the MAC aborted the frame because 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. †

Transmit Descriptor 1 (TDES1)

TDES1 contains the buffer sizes and other bits which control the descriptor chain or ring and the frame being transferred. †

Refer to "Buffer Size Calculations" on page 17–24 for further detail on calculating buffer sizes. †

Table 17–7. Transmit Descriptor 1 (Part 1 of 2)

Bit	Description
31	IC: Interrupt on Completion When set, this bit sets Transmit Interrupt (Register 5[0]) after the present frame has been transmitted. †
30	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 should have a non-zero value. †
29	FS: First Segment When set, this bit indicates that the buffer contains the first segment of a frame. †
28:27	CIC: Checksum Insertion Control These bits control the insertion of checksums in Ethernet frames that encapsulate TCP, UDP, or ICMP over IPv4 or IPv6 as described below. ■ 0: Do nothing. Checksum Engine is bypassed ■ 1: Insert IPv4 header checksum. Use this value to insert IPv4 header checksum when the frame encapsulates an IPv4 datagram. ■ 2: Insert TCP/UDP/ICMP checksum. The checksum is calculated over the TCP, UDP, or ICMP segment only and the TCP, UDP, or ICMP pseudo-header checksum is assumed to be present in the corresponding input frame's Checksum field. An IPv4 header checksum is also inserted if the encapsulated datagram conforms to IPv4. ■ 3: Insert a TCP/UDP/ICMP checksum that is fully calculated in this engine. In other words, the TCP, UDP, or ICMP pseudo-header is included in the checksum calculation, and the input frame's corresponding Checksum field has an all-zero value. An IPv4 Header checksum is also inserted if the encapsulated datagram conforms to IPv4. The Checksum engine detects whether the TCP, UDP, or ICMP segment is encapsulated in IPv4 or IPv6 and processes its data accordingly.

Table 17–7. Transmit Descriptor 1 (Part 2 of 2)

Bit	Description
26	DC: Disable CRC When set, the MAC does not append the CRC to the end of the transmitted frame. This is valid only when the first segment (TDES1[29]) is set. †
25	TER: Transmit End of Ring When set, this bit indicates that the descriptor list reached its final descriptor. The returns to the base address of the list, creating a descriptor ring. †
24	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 TDES1[24] is set, TBS2 (TDES1[21–11]) are “don’t care” values. TDES1[25] takes precedence over TDES1[24]. †
23	DP: Disable Padding When set, the MAC does not automatically add padding to a frame shorter than 64 bytes. When this bit is reset, 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 (TDES1[26]) bit. This is valid only when the first segment (TDES1[29]) is set. †
22	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 (TDES1[29]) is set. †
21:11	TBS2: Transmit Buffer 2 Size These bits indicate the second Data Buffer in bytes. This field is not valid if TDES1[24] is set. †
10:0	TBS1: Transmit Buffer 1 Size These bits indicate the First Data Buffer byte size. If this field is 0, the DMA ignores this buffer and uses Buffer 2 or next descriptor depending on the value of TCH (Bit 24). †

Transmit Descriptor 2 (TDES2)

TDES2 contains the address pointer to the first buffer of the descriptor. †

Table 17–8. Transmit Descriptor 2

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. Refer to “Host Data Buffer Alignment” on page 17–24 for further detail on buffer address alignment. †

Transmit Descriptor 3 (TDES3)

TDES3 contains the address pointer either to the second buffer of the descriptor or the next descriptor. †

Table 17–9. Transmit Descriptor 3

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 (TDES1[24]) 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 TDES1[24] is set. (LSBs are ignored internally.) †

Receive Descriptor

The DMA in the EMAC requires at least two descriptors when receiving a frame. The receive state machine of the DMA always attempts to acquire an extra descriptor in anticipation of an incoming frame. (The size of the incoming frame is unknown). Before the RX DMA closes a descriptor, it attempts to acquire the next descriptor even if no frames are received.

In a single descriptor (receive) system, the subsystem generates a descriptor error if receive buffer is unable to accommodate the incoming frame and the next descriptor is not owned by the DMA. Thus, the Host is forced to increase either its descriptor pool or the buffer size. Otherwise, the subsystem starts dropping all incoming frames. †

Receive Descriptor 0 (RDES0)

RDES0 contains the received frame status, the frame length, and the descriptor ownership information.

Table 17–10. Receive Descriptor 0 (Part 1 of 3)

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 reset, 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 bit is reset. 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. †
15	ES: Error Summary Indicates the logical OR of the following bits: ■ RDES0[0]: Payload Checksum Error ■ RDES0[1]: CRC Error ■ RDES0[3]: Receive Error ■ RDES0[4]: Watchdog Timeout ■ RDES0[6]: Late Collision ■ RDES0[7]: IPC Checksum (Type 2) ■ 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 field is valid only when the Last Descriptor (RDES0[8]) is set. †

Table 17–10. Receive Descriptor 0 (Part 2 of 3)

Bit	Description
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 reset. Length error status is not valid when CRC error is present. †
11	OE: Overflow Error When set, this bit indicates that the received frame is damaged because of buffer overflow in RX FIFO buffer. Note: This bit is set only when the DMA transfers a partial frame to the application. This 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 RX FIFO buffer. †
10	VLAN: VLAN Tag When set, this bit indicates that the frame pointed to by this descriptor is a VLAN frame tagged by the MAC. The VLAN tagging depends on checking VLAN fields of the received frame based on the Register 7 (VLAN Tag Register) settings. †
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	LS: Last Descriptor When set, this bit indicates that the buffers pointed to by this descriptor are the last buffers of the frame †
7	IPC Checksum Error When IP Checksum Engine (Type 1) is enabled, this bit, when set, indicates that the 16-bit IPv4 Header checksum calculated by the MAC did not match the received checksum bytes. Bit 15 (ES) is NOT set when this bit is set in this mode. If this bit is set when Full Checksum Offload Engine (Type 2) is enabled, it indicates an error in the IPv4 or IPv6 header. This error can be due to inconsistent Ethernet Type field and IP header Version field values, a header checksum mismatch in IPv4, or an Ethernet frame lacking the expected number of IP header bytes. For more information, refer to Table 17–11 on page 17–42.
6	LC: Late Collision When set, this bit indicates that a late collision has occurred while receiving the frame in 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 reset, it indicates that the received frame is an IEEE802.3 frame. This bit is not valid for Runt frames less than 14 bytes. For more information, refer to Table 17–11 on page 17–42. †
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. †
3	RE: Receive Error When set, this bit indicates that the `gmii_rxer_i` signal is asserted while `gmii_rxdv_i` is asserted during frame reception. Error can be of less or no extension, or error (rxd != 0xf) during extension. †
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 MII Mode. †

Table 17–10. Receive Descriptor 0 (Part 3 of 3)

Bit	Description
1	CE: CRC Error When set, this bit indicates that a CRC error occurred on the received frame. This field is valid only when the Last Descriptor (RDES0[8]) is set. †
0	RX MAC Address or Payload Checksum Error When set, this bit indicates that the RX MAC Address registers value (1 to 15) matched the frame's DA field. When reset, this bit indicates that the RX MAC Address Register 0 value matched the DA field. If Full Checksum Offload Engine is enabled, this bit, when set, indicates the TCP, UDP, or ICMP checksum the EMAC calculated does not match the received encapsulated TCP, UDP, or ICMP segment's Checksum field. This bit is also set when the received number of payload bytes does not match the value indicated in the Length field of the encapsulated IPv4 or IPv6 datagram in the received Ethernet frame. For more information, refer to Table 17–11 on page 17–42. †

When the Full Checksum Offload Engine (Type 2) is enabled, the permutations of Bits 5, 7, and 0 reflect the conditions discussed in Table 17–11. †

Table 17–11. Receive Descriptor 0 when COE (Type 2) is Enabled

Bit 5: Frame Type	Bit 7: IPC Checksum Error	Bit 0: Payload Checksum Error	Frame Status
0	0	0	IEEE 802.3 Type frame (Length field value is less than 0x0600.) †
1	0	0	IPv4/IPv6 Type frame, no checksum error detected †
1	0	1	IPv4/IPv6 Type frame with a payload checksum error (as described for PCE) detected †
1	1	0	IPv4/IPv6 Type frame with an IP header checksum error (as described for IPC CE) detected †
1	1	1	IPv4/IPv6 Type frame with both IP header and payload checksum errors detected †
0	0	1	IPv4/IPv6 Type frame with no IP header checksum error and the payload check bypassed, due to an unsupported payload †
0	1	1	A Type frame that is neither IPv4 or IPv6 (the Checksum Offload engine bypasses checksum completely.) †
0	1	0	Reserved †

Receive Descriptor 1 (RDES1)

RDES1 contains the buffer sizes and other bits that control the descriptor chain or ring. †

For more information about calculating buffer sizes, refer to "Buffer Size Calculations" on page 17–24. †

Table 17–12. Receive Descriptor 1

Bit	Description
31	Disable Interrupt on Completion When set, this bit prevents the setting of the bit 6 (RI) of the Register 5 (Status Register) for the received frame that ends in the buffer pointed to by this descriptor. This, in turn, disables the assertion of the interrupt to Host due to RI for that frame. †
30:26	Reserved †
25	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. †
24	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 RDES1[24] is set, RBS2 (RDES1[21-11]) is a "don't care" value. RDES1[25] takes precedence over RDES1[24]. †
23:22	Reserved †
21:11	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) is not aligned to bus width. In the case where the buffer size is not a multiple of 4, the resulting behavior is undefined. This field is not valid if RDES1[24] is set.
10:0	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) is not aligned. In the case where 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 next descriptor depending on the value of RCH (Bit 24).

Receive Descriptor 2 (RDES2)

RDES2 contains the address pointer to the first data buffer in the descriptor. †

For more information about buffer address alignment, refer to"Host Data Buffer Alignment" on page 17–24. †

Table 17–13. Receive Descriptor 2 (Default Operation)

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 frame. The DMA performs a write operation with the RDES2[1:0] bits as 0 during the transfer of the start of frame but the frame data 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.

Receive Descriptor 3 (RDES3)

RDES3 contains the address pointer either to the second data buffer in the descriptor or to the next descriptor. †

Table 17–14. Receive Descriptor 3

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[24]) bit is set, this address contains the pointer to the physical memory where the Next descriptor is present. If RDES1[24] is set, the buffer (Next descriptor) address pointer must be bus width-aligned (RDES3[1:0] = 0. LSBs are ignored internally.) However, when RDES1[24] is reset, 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.

Descriptor Format With IEEE 1588-2005 Timestamps Enabled

The default descriptor format (as described in "Transmit Descriptor" on page 17–36 and "Receive Descriptor" on page 17–40), and field descriptions remain unchanged when created by software (Own bit is set in DES0). †

However, if the software has enabled IEEE 1588-2005 functionality, the DES2 and DES3 descriptor fields (refer to Table 17–15) take on a different meaning when the DMA closes the descriptor (Own bit in DES0 is cleared). †

The DMA updates the DES2 and DES3 with the timestamp value before clearing the Own bit in DES0. When the EMAC is operating in 32-bit mode, DES2 is updated with the lower 32 timestamp bits (the Sub-second field, called TSL in subsequent sections) and DES3 is updated with the upper 32 timestamp bits (the seconds field, called TSH in subsequent sections).

Table 17–15. Descriptor Fields when DMA Clears the Own Bit

	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
DES0																																
DES1																																
DES2	Timestamp Low [31:0]																															
DES3	Timestamp High [31:0]																															

The following sections describe the details specific to receive and transmit descriptors in this mode. †

Transmit Descriptor

In addition to the changes described in "Descriptor Format With IEEE 1588-2005 Timestamps Enabled" on page 17–44, the transmit descriptor has additional control and status bits (TTSE and TTSS, respectively) for timestamping, as shown in Table 17–16. Software sets the TTSE bit (when the Own bit is set), instructing the EMAC to generate a timestamp for the corresponding Ethernet frame being transmitted. The DMA sets the TTSS bit if the timestamp has been updated in the TDES2 and TDES3 fields when the descriptor is closed (Own bit is cleared).

Table 17–16. Transmit Descriptor Fields - Normal 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	RES													TTSS	Status [16:0]																
TDES1	Other Control Fields									TTSE	Buffer 2Byte Count [21:11]											Buffer 1 Byte Count [10:0]									
TDES2	TTSL																															
TDES3	TTSH																															

Transmit Timestamp Control and Status Fields

The position of these fields is different for normal transmit descriptor and enhanced format transmit descriptor. The value of this field in both the cases shall be preserved by the DMA at the time of closing the descriptor. †

Updates to Table 17–14 on page 17–44 and Table 17–6 on page 17–36 for the default (normal) descriptor format are described below. †

Table 17–17. Transmit Timestamp Status – Normal Descriptor Format Case (TDES0)

Bit	Description
17	TTSS: Transmit Timestamp Status This field is a status bit indicating that a timestamp was captured for the corresponding transmit frame. When this bit is set, both TDES2 and TDES3 have a timestamp value that was captured for the transmit frame. This field is valid only when the Last Segment control bit (TDES1[30] in the descriptor) is set. †

Table 17–18. Transmit Timestamp Control – Normal Descriptor Format Case (TDES1)

Bit	Description
22	TTSE: Transmit Timestamp Enable When set, this field enables IEEE1588 hardware timestamping for the transmit frame described by the descriptor. This field is valid only when the First Segment control bit (TDES1[29] in the descriptor) is set. †

Transmit Timestamp Field

The transmit descriptor format and field descriptions remain unchanged when they are created by software (when the Own bit is set). However, when the DMA closes the last descriptor (marked, in the alternative descriptor format, by the LS bit in TDES1 or TDES0) and IEEE 1588 functionality is enabled (the Own bit is cleared), the TDES2 and TDES3 descriptor fields are updated with the timestamp, if taken, for that frame. †

Table 17–19 and Table 17–20 describe the fields that have different meaning when the descriptor is closed. †

Table 17–19. Transmit Descriptor Fields (TDES2)

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 control bit (LS) in the descriptor is set. †

Table 17–20. Transmit Descriptor Fields (TDES3)

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 transmit frame. This field has the timestamp only if the Last Segment control bit (LS) in the descriptor is set. †

Receive descriptor

Receive Timestamp

Table 17–21 shows the format of receive descriptor when timestamping is enabled.

Table 17–21. Receive Timestamp Fields

	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																																
RDES1																																
RDES2	Receive Frame Timestamp Low [31:0]																															
RDES3	Receive Frame Timestamp High [31:0]																															

Table 17–22 on page 17–47 and Table 17–23 on page 17–47 describe the fields that have different meaning for RDES2 and RDES3 when the receive descriptor is closed and timestamping is enabled.

When software disables the Timestamp feature (the TSENA bit in Register 448 is low), the DMA does not update the RDES2 or RDES3 fields of descriptor before closing the RDES0. †

Table 17–22. Receive Descriptor Fields (RDES2)

Bit	Description
31:0	RTSL: Receive Frame Timestamp Low The DMA updates this field with the least significant 32 bits of the timestamp captured for the corresponding receive frame. The DMA updates this field only for the last descriptor of the receive frame indicated by Last Descriptor status bit (RDES0[8]). When this field and the RTSH field in RDES3 show an all-ones value, the timestamp must be treated as corrupt. †

Table 17–23. Receive descriptor Fields (RDES3)

Bit	Description
31:0	RTSH: Receive Frame Timestamp High The DMA updates this field with the most significant 32 bits of the timestamp captured for the corresponding receive frame. The DMA updates this field only for the last descriptor of the receive frame indicated by Last Descriptor status bit (RDES0[8]). When this field and RDES2's RTSL field show all-ones values, the timestamp must be treated as corrupt. †

Alternate or Enhanced Descriptors

The alternate (or enhanced) descriptor structure can have 8 DWORDS (32-bytes) instead of the 4 DWORDS as in the case of normal descriptor format. The features of the alternate descriptor structure are:

- The normal descriptor structure allows data buffers of up to 2,048 bytes. The alternate 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. †
- This 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 needs to allocate 32-bytes (8 DWORDS) of memory for every descriptor. For this, the software should set Bit 7 (Alternate 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, the software can use alternate descriptors with the default size of 16 bytes (4 DWORDS). For this, the software should reset Bit 7 (Alternate Descriptor Size) of Register 0 (Bus Mode Register) to 0. †
- When alternate descriptor is selected without Timestamp or Receive IPC Full Checksum Offload Engine (Type 2) feature, the descriptor size is always 4 DWORDs (DES0-DES3). Therefore, the software can use alternate descriptors with the default size of 16 bytes. †

Transmit Descriptor

The transmit descriptor structure is shown in Table 17–24 on page 17–49. The application software must program the control bits TDES0[31:18] during descriptor initialization. When the DMA updates the descriptor, it write backs all the control bits except the OWN bit (which it clears) and updates the status bits[7:0]. The contents of the transmitter descriptor word 0 (TDES0) through word 3 (TDES3) are given in Table 17–26 on page 17–50, Table 17–27 on page 17–53, Table 17–28 on page 17–53, and Table 17–29 on page 17–53, respectively. †

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. This is indicated by the status Bit 17 (TTSS) of TDES0 shown in Table 17–24 on page 17–49. The contents of TDES6 and TDES7 are mentioned in Table 17–30 on page 17–53 and Table 17–31 on page 17–53. †

When advanced timestamp feature is enabled, the 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 reset, the TDES4-TDES7 descriptor space is not valid. †

Table 17–24. Transmit Descriptor Fields - Alternate (Enhanced) 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]					TTSE	Ctrl [24:18]							TTSS	Status [16:7]										Ctrl/Status [6:3]				Status [2:0]			
TDES1	Ctrl [31:29]	Buffer 2Byte 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 as shown in Table 17–25. †

Table 17–25. Transmit Descriptor Fetch (Read) for Alternate (Enhanced) 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]					TTSE	Ctrl [24:18]							Reserved for Status [17:7]											SLOT Number [6:3]				Reserved for Status [2:0]		
TDES1	Ctrl [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 17–26. Transmit Descriptor Word 0 (TDES0) (Part 1 of 3)

Bit	Description
31	OWN: Own Bit When set, this bit indicates that the descriptor is owned by the DMA. When this bit is reset, 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. This avoids a possible race condition between fetching a descriptor and the driver setting an ownership bit. †
30	IC: Interrupt on Completion When set, this bit sets the Transmit Interrupt (Register 5[0]) 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 is valid only when the first segment (TDES0[28]) is set. †
26	DP: Disable Pad When set, the MAC does not automatically add padding to a frame shorter than 64 bytes. When this bit is reset, 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 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

Table 17–26. Transmit Descriptor Word 0 (TDES0) (Part 2 of 3)

Bit	Description
23:22	CIC: Checksum Insertion Control These bits control the checksum calculation and insertion. The following list describes the bit encoding: ■ 0: Checksum Insertion Disabled. ■ 1: Only IP header checksum calculation and insertion are enabled. ■ 2: IP header checksum and payload checksum calculation and insertion are enabled, but pseudoheader checksum is not calculated in hardware. ■ 3: 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. †
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 IP header 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. †
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 †

Table 17–26. Transmit Descriptor Word 0 (TDES0) (Part 3 of 3)

Bit	Description
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. †
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 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 form the PHY was not asserted during transmission. †
9	Reserved
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) bit in the 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 high. †
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 17–27. 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. For more information about calculating buffer sizes, refer to "Buffer Size Calculations" on page 17–24. †
15:13	Reserved †
12:0	TBS1: Transmit Buffer 1 Size These bits indicate 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 17–28. 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. For more information about buffer address alignment, refer to "Host Data Buffer Alignment" on page 17–24. †

Table 17–29. 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 (TDES1[24]) 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 TDES1[24] is set. (LSBs are ignored internally.) †

Table 17–30. 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 17–31. 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 structure of the received descriptor is shown in Table 17–32. This 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, the 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 reset, the RDES0[0] is always cleared and the RDES4-RDES7 descriptor space is not valid. †

Table 17–32. Receive Descriptor Fields - Alternate (Enhanced) 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	Status [30:0]																														
TDES1	CTRL	RES [30:29]	Buffer 2Byte Count [28:16]														Ctrl [15:14]		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	Extended status [31:0]																															
TDES5	Reserved																															
TDES6	Transmit Timestamp Low [31:0]																															
TDES7	Transmit Timestamp High [31:0]																															

The contents of RDES0 are identified in Table 17–33. The contents of RDES1 through RDES3 are identified in Table 17–34 on page 17–56, Table 17–35 on page 17–57, and Table 17–36 on page 17–57, respectively. †

Table 17–33. Receive Descriptor Fields (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 reset, 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. †

Table 17–33. Receive Descriptor Fields (RDES0)

Bit	Description
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 reset. 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. †
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 ■ 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 field is valid only when the Last Descriptor (RDES0[8]) 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 reset. †
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. This 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 RX FIFO buffer. †
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 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	LS: Last Descriptor When set, this bit indicates that the buffers pointed to by this descriptor are the last buffers of the frame †

Table 17–33. Receive Descriptor Fields (RDES0)

Bit	Description
7	Timestamp Available, IP Checksum Error (Type1), or Giant Frame When advanced timestamp feature is present, when set, this bit 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. When IP Checksum Engine (Type 1) is selected, this bit, when set, indicates that the 16-bit IPv4 Header checksum calculated by the EMAC did not match the received checksum bytes. Otherwise, this bit, when set, indicates the Giant frame Status. Giant frames are larger than 1,518-byte (or 1,522-byte for VLAN or 2,000-byte when Bit 27 (2KPE) of MAC Configuration register is set) normal frames and larger than 9,018-byte (9,022-byte for VLAN) frame when Jumbo frame processing is enabled.
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 reset, 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. †
3	RE: Receive Error When set, this bit indicates that the `gmii_rxer_i` signal is asserted while `gmii_rxdv_i` is asserted during frame reception. Error can be of less or no extension, or error (rxd !=0xf) during extension. †
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 field 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 is valid only when the Last Descriptor bit (RDES0[8]) is set. When 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 reset, this bit indicates that the RX MAC Address Register 0 value matched the DA field. †

Table 17–34. Receive Descriptor Fields 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. This, in turn, disables the assertion of the interrupt to Host because of RI for that frame. †
30:29	Reserved †

Table 17–34. Receive Descriptor Fields 1 (RDES1)

Bit	Description
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) is not aligned to 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 "Buffer Size Calculations" on page 17–24.
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) 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 next descriptor depending on the value of RCH (Bit 14). For more information about calculating buffer sizes, refer to "Buffer Size Calculations" on page 17–24.

Table 17–35. Receive Descriptor Fields 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 "Host Data Buffer Alignment" on page 17–24.

Table 17–36. Receive descriptor Fields 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[24]) bit is set, this address contains the pointer to the physical memory where the Next descriptor is present. If RDES1[24] is set, the buffer (Next descriptor) address pointer must be bus width-aligned (RDES3[1:0] = 0. LSBs are ignored internally.) However, when RDES1[24] is reset, 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.

The extended status written is as shown in Table 17–37. 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 of RDES0. This status is available only when the Advance Timestamp or IPC Full Offload feature is selected. †

Table 17–37. Receive Descriptor Fields 4 (RDES4) (Part 1 of 2)

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 high. 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 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.
13	PTP Version When set, this bit indicates that the received PTP message is having the IEEE 1588 version 2 format. When reset, 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.

Table 17–37. Receive Descriptor Fields 4 (RDES4) (Part 2 of 2)

Bit	Description
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.
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. ■ 0: Unknown or did not process IP payload ■ 1: UDP ■ 2: TCP ■ 3: ICMP ■ 4–7: Reserved This bit is valid when either Bit 7 or Bit 6 is set.

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. The contents of RDES6 and RDES7 are identified in Table 17–38 and Table 17–39 on page 17–60, respectively. †

Table 17–38. Receive Descriptor Fields 6 (RDES6)

Bit	Description
31:0	RTSL: Receive Frame Timestamp Low This field is updated by DMA with the least significant 32 bits of the timestamp captured for the corresponding receive frame. This field is updated by DMA only for the last descriptor of the receive frame which is indicated by Last Descriptor status bit (RDES0[8]). †

Table 17–39. Receive Descriptor Fields 7 (RDES7)

Bit	Description
31:0	RTSH: Receive 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 is updated by DMA only for the last descriptor of the receive frame which is indicated by Last Descriptor status bit (RDES0[8]). †

Initializing DMA

This section provides the instructions for initializing the DMA/MAC registers in the proper sequence. Perform the following steps to initialize the DMA:

1. Provide a software reset. This resets 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 (AHB or AXI Status) to confirm that all previously initiated (before software-reset) or ongoing transactions are complete.

   ☞ 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) 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): †
   a. Mixed Burst and AAL
   b. Fixed burst or undefined burst †
   c. Burst length values and burst mode values. †
   d. Descriptor Length (only valid if Ring Mode is used) †
   e. TX and RX DMA Arbitration scheme †
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]). †

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. For more information about descriptors, refer to"Normal Descriptor" on page 17–36 and "Alternate or Enhanced Descriptors" on page 17–47. †
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)
   a. Receive and Transmit Store And Forward †
   b. Receive and Transmit Threshold Control (RTC and TTC) †
   c. Hardware Flow Control enable †
   d. Flow Control Activation and De-activation thresholds for MTL Receive and Transmit FIFO buffers (RFA and RFD) †
   e. Error frame and undersized good frame forwarding enable †
   f. 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 the Register 7 (Interrupt Enable Register). †

    Perform step 12 only if you did not perform step 3. †

12. Read Register 11 (AHB or AXI Status) to confirm that all previous transactions are complete. †

    If any previous transaction is still in progress when you read the Register 11 (AHB or 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)). †

Initializing MAC

The following MAC Initialization operations can be performed after DMA initialization. If the MAC initialization is done before the DMA is set-up, then enable the MAC receiver (last step below) only after the DMA is active. Otherwise, received frames fills the RX FIFO buffer and overflow. †

1. Program the EMAC Register 4 (GMII Address Register) for controlling the management cycles for external PHY. For example, Physical Layer Address PA (bits 15-11). In addition, set bit 0 (GMII Busy) for writing into PHY and reading from PHY. †
2. Read the 16-bit data of Register 5 (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 Register 4 (GMII Address Register). †
3. Provide the MAC address registers (Register 16 (MAC Address0 High Register) and Register 17 (MAC Address0 Low Register)). Because 128 MAC addresses are supported, you need to program the MAC addresses accordingly.
4. Program Register 2 (Hash Table High Register) and Register 3 (Hash Table Low Register).
5. Program the following fields to set the appropriate filters for the incoming frames in Register 1 (MAC Frame Filter): †
   a. Receive All †
   b. Promiscuous mode †
   c. Hash or Perfect Filter †
   d. Unicast, multicast, broadcast, and control frames filter settings †
6. Program the following fields for proper flow control in Register 6 (Flow Control Register): †
   a. Pause time and other pause frame control bits †
   b. Receive and Transmit Flow control bits †
   c. 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 Register 0 (MAC Configuration Register). For example, Interframe gap while transmission and jabber disable. Based on the Auto-negotiation you can set the Duplex mode (bit 11) or port select (bit 15). †
9. Set Bit 3 (TE) and Bit 2 (RE) in Register 0 (MAC Configuration Register). †

Do not change the configuration (such as duplex mode, speed, port, or loopback) when the EMAC DMA is actively transmitting or receiving. The 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 register (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 MAC transmitter and MAC 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 TX FIFO buffer and RX FIFO buffer are empty. †
6. To re-start the operation, first start the DMAs, and then enable the MAC 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 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 chip 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). †

   ☞ 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). †

   ☞ To switch off the transmit clock during the LPI state, use the `sbd_tx_clk_gating_ctrl_o` signal for gating the clock input. †

   ☞ 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 the step 6 when you want to come out of the LPI state. †

6. Reset 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. †

For restoring 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, reset 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 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). This instructs the MAC to use the Target Time registers (register 455 and 456) for 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. This instructs the MAC to generate 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 after the time at which the previous pulse is de-asserted. This instructs the MAC to generate single pulse on the PPS signal output, 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 interrupt) in Bits [6:5], TRGTMODSEL, of Register 459 (PPS Control Register). This instructs the MAC to use the Target Time registers (register 455 and 456) for 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. This instructs the MAC to generate train of pulses on the PPS signal output with start time programmed in 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 reset 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. This stops the train of pulses on PPS signal output after the programmed stop time specified in Step 6 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) by programming 0110 in the PPSCMD field before the time (programmed in step 6) elapses. You can cancel the pulse train generation by programming 0011 in the PPSCMD field before the programmed start time (in step 2) elapses. †

Generating an Interrupt without Affecting the PPS

The Bits [6:5], TRGTMODSEL, of the 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 event: †

1. Program 00 (for interrupt) in Bits [6:5], TRGTMODSEL, of Register 459 (PPS Control Register). This instructs the MAC to use the Target Time registers (register 455 and 456) for target time interrupt. †
2. Program a target time value in the Target Time registers (register 455 and 456). This instructs 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 over-written with the new mode and new programmed Target Time register value.

Ethernet MAC Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for either of the following module instances:

- **emac0**
- **emac1**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 17–40 shows the revision history for this document.

Table 17–40. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	Added programming model section.
January 2012	1.0	Initial release.

18. USB 2.0 OTG Controller

cv_54018-1.2

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

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 (www.usb.org).

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
- Supports all USB 2.0 speeds:
  - High speed (HS, 480-Mbps)
  - Full speed (FS, 12-Mbps)
  - Low speed (LS, 1.5-Mbps)

  ☞ In host mode, all speeds are supported. However, in device mode, only high speed and full speed are supported.

- 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

  ☞ 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

  ☞ 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:

- 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

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

Table 18–1 lists some PHYs that are compatible with the USB OTG.

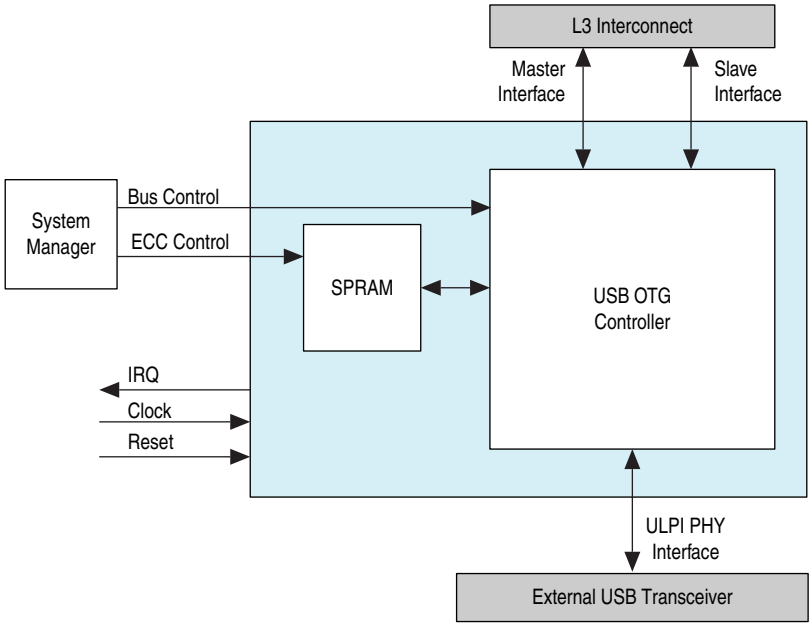
Table 18–1. Supported PHYs

Manufacturer	Part Number
TI	TUSB1210
NXP	ISP1504
Cypress	CY7C68003
SMSC	USB3300

USB OTG Controller Block Diagram and System Integration

Figure 18–1 is a block diagram showing one USB OTG controller subsystem in the HPS. Two subsystems are included in the HPS.

Figure 18–1. USB OTG Controller System Integration



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.

For more information, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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:

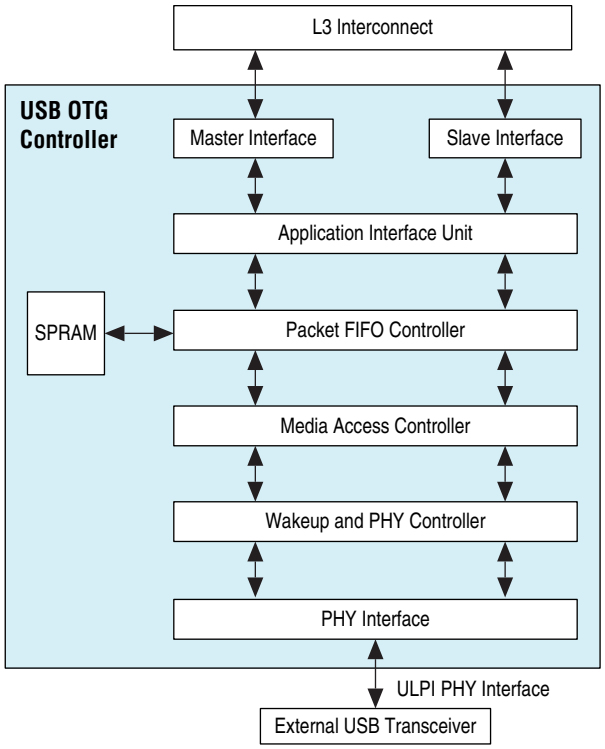
- Clock input from the clock manager to the USB OTG controller
- Reset input from the reset manager to the USB OTG controller
- Interrupt line from the USB OTG controller to the microprocessor unit (MPU) global interrupt controller (GIC).

Functional Description of the USB OTG Controller

USB OTG Controller Block Description

Figure 18–2 is a block diagram of the USB OTG controller. The following sections provide detail about each of the units that comprise the USB OTG controller.

Figure 18–2. USB OTG Controller Block Diagram



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.

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 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
- Link power management (LPM) functions

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.

LPM Functions

The USB OTG controller supports LPM in both host and device modes. With this feature, the USB OTG controller can enter a sleep state when a successful LPM transaction occurs on the USB link.

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.

ULPI PHY Interface

The ULPI PHY interface is synchronous to the ulpi_clk signal coming from the PHY. Table 18–2 lists the ULPI PHY interface names and related information.

Table 18–2. ULPI PHY Interfaces

Port Name	Bit Width	Direction	Description
ulpi_clk	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.
ulpi_dir	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.
ulpi_nxt	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.
ulpi_stp	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.
ulpi_data[7:0]	8	Bidirectional	Bidirectional data bus. Driven low by the controller during idle.

Clocks

All clocks must be operational when reset is released. No special handling is required on the clocks.

Table 18–3 lists the USB OTG controller clock inputs.

Table 18–3. USB OTG Controller Clock Inputs

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.

Interrupts

Each USB OTG controller has a single interrupt output. Interrupts are asserted on the conditions shown in Table 18–4.

Table 18–4. USB OTG Interrupt Conditions

Condition	Mode
Device-initiated remote wakeup is detected.	Host mode
Session request is detected from the device.	Host mode
Device disconnect is detected.	Host mode
Host LPM entry retry has expired or LPM transaction(s) are complete.	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
LPM handshake is sent.	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
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 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 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.

The `directfifo` memory space is described in the controller address map. Refer to "USB OTG Controller Address Map and Register Definitions" on page 18–15.

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.

Host Operation

Host Initialization

After power up, the USB port is in its default mode. No V_{BUS} 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 (`prtpwr`) bit to 1 in the Host Port Control and Status Register (`hprt`) 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 `hprt` is set to 1.
3. Upon detecting a port connection, the software driver initiates a port reset by setting the Port Reset (`prtrst`) bit to 1 in `hprt`.
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 `prtrst` back to 0 to release the port reset.

6. The USB OTG controller generates an interrupt. The Port Enable Disable Change (`prtenchng`) and Port Speed (`prtspd`) bits, in `hprt`, 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 (`hfir`) is updated with the corresponding PHY clock settings. The `hfir`, 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 (`grxfsiz`)—selects the size of the receive FIFO buffer

   b. Non-periodic Transmit FIFO Size Register (`gnptxfsiz`)—selects the size and the start address of the non-periodic transmit FIFO buffer for nonperiodic transactions

   c. Host Periodic Transmit FIFO Size Register (`hptxfsiz`)—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 will access 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.

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.

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 OTG Controller Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for either of the following module instances:

- **usb0**
- **usb1**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 18–5 shows the revision history for this document.

Table 18–5. Document Revision History

Date	Version	Changes
November 2012	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.
June 2012	1.1	Added information about ECCs.
January 2012	1.0	Initial release.

19. SPI Controller

cv_54019-1.2

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).

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 `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

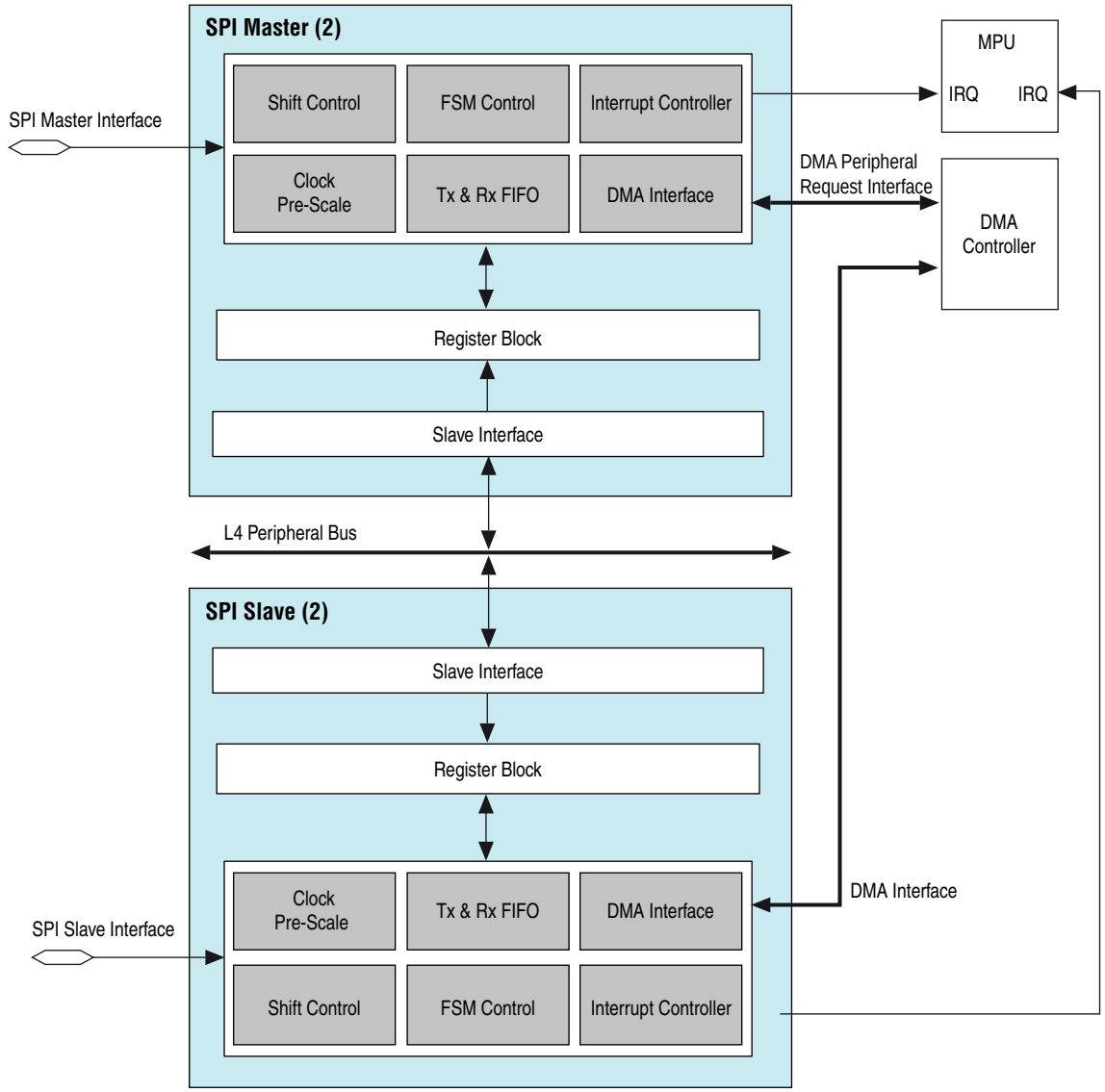
SPI Block Diagram and System Integration

The SPI supports data bus widths of 32 bits. †

SPI Block Diagram

Figure 19–1 shows the functional groupings of the main interfaces to the SPI block. †

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

Functional Description of the SPI Controller

Protocol Details and Standards Compliance

This chapter 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: †

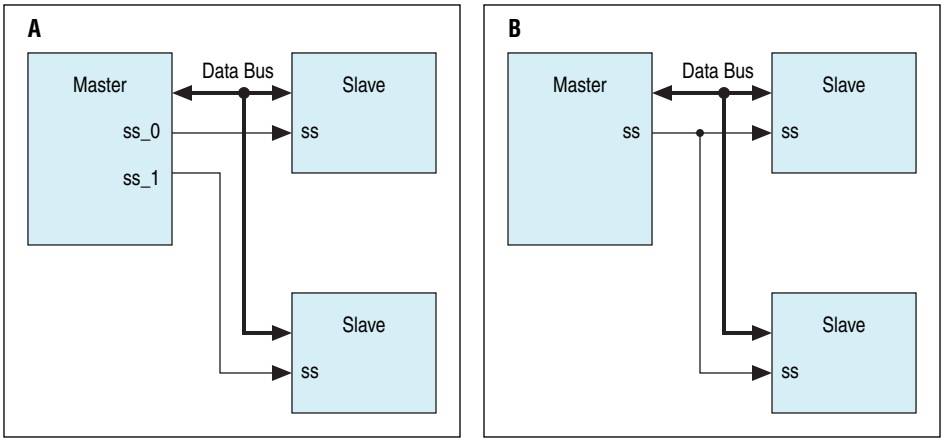
- Motorola SPI protocol – A four-wire, full-duplex serial protocol from Motorola. The slave select line is held high when the SPI controller is idle or disabled. For more information, refer to "Motorola SPI Protocol" on page 19–13. †
- Texas Instruments Serial Protocol (SSP) – A four-wire, full-duplex serial protocol. The slave select line used for SPI and Microwire protocols doubles as the frame indicator for the SSP protocol. For more information, refer to "Texas Instruments Synchronous Serial Protocol (SSP)" on page 19–14. †
- 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" on page 19–15. 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 Figure 19–2. Software would control which slave is to respond to the serial transfer request from the master device. †

Figure 19–2. Hardware/Software Slave Selection †

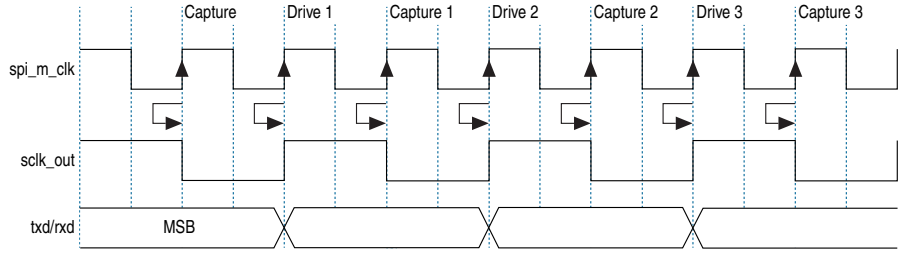


Serial Bit-Rate Clocks

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, as shown in Figure 19–3. 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 Equation 19–1, where <*SPI clock*> is spi_m_clk for the master SPI modules and l4_main_clk for the slave SPI modules. †

Equation 19–1.

$$F_{sclk\_out} = F_{<SPI\ clock>} / SCKDV$$

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. †

Equation 19–2 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. †

Equation 19–2. SPI Master Peripheral Clock

$$F_{spi\_m\_clk} >= 2 \times (\text{maximum } F_{sclk\_out}) \dagger$$

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 rxd 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): $F_{l4\_main\_clk} >= 6 \times (\text{maximum } F_{sclk\_in})$ †
- Slave: $F_{l4\_main\_clk} >= 8 \times (\text{maximum } F_{sclk\_in})$ †

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. For detailed information about reset signals, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone® V Device Handbook.*

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 Overrun 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. †

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 in the modes discussed in this section.

The transfer mode (TMOD) is set by writing to control register 0 (CTRLR0). †

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

This transfer mode is only valid for serial masters. †

When TMOD = 3, the transmit data is used to transmit an opcode and/or an address to the EEPROM device. Typically this takes three data frames (8-bit opcode followed by 8-bit upper address and 8-bit lower address). During the transmission of the opcode and address, no data is captured by the receive logic (as long as the 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. Therefore, 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. †

☞ 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. Figure 19–1 on page 19–2 shows a SPI master. †

The serial bit-rate clock, generated and controlled by the SPI controller, is driven out on the sclk_out line. When the SPI controller is disabled, no serial transfers can occur and sclk_out is held in "inactive" state, as defined by the serial protocol under which it operates. †

RXD Sample Delay

SPI master device is capable of delaying the default sample time of the rxd 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 rxd signal from the slave can mean that the timing of the rxd signal, as seen by the master, has moved away from the normal sampling time.

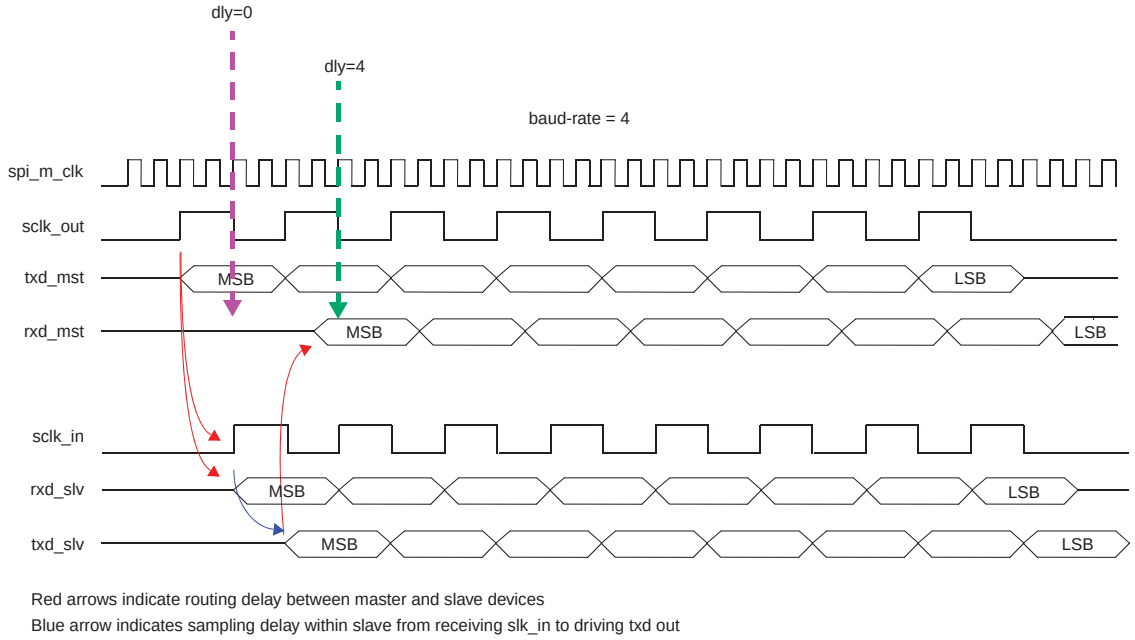
Without the RXD sample delay, you must increase the baud rate for the transfer in order to ensure that the setup times on the rxd signal are within range. This reduces the frequency of the serial interface.

Additional logic is included in the SPI master to delay the default sample time of the rxd signal. This additional logic can help to increase the maximum achievable frequency on the serial bus. †

By writing to the rsd field of the RX Sample Delay Register (rx_sample_dly), you specify an additional amount of delay applied to the rxd sample, in number of spi_m_clk clock cycles, up to 64 cycles. If the rsd field is programmed with a value exceeding 64, zero delay is applied to the rxd sample.

Round trip routing delays on the sclk_out signal from the master and the rxd signal from the slave can mean that the timing of the rxd signal, as seen by the master, has moved away from the normal sampling time. Figure 19–4 illustrates this situation. Red arrows indicate routing delay between master and slave devices. Blue arrow indicates sampling delay within slave from receiving sclk_in to driving txd out. †

Figure 19–4. Effects of Round-Trip Routing Delays on sclk_out Signal



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. †

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 Transit FIFO Empty (TFE) status (waiting for 1) or wait for (BAUDR * SPI clock) clock cycles. †

Master SPI and SSP Serial Transfers

"Motorola SPI Protocol" on page 19–13 and "Texas Instruments Synchronous Serial Protocol (SSP)" on page 19–14 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 and/or address 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). †

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. †

Master Microwire Serial Transfers

"National Semiconductor Microwire Protocol" on page 19–15 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*. †

SPI Slave

The SPI slave handles serial communication with transfer initiated and controlled by serial master peripheral devices.

Figure 19–5 shows an example of a SPI slave in a single-master bus system, including the following signals: †

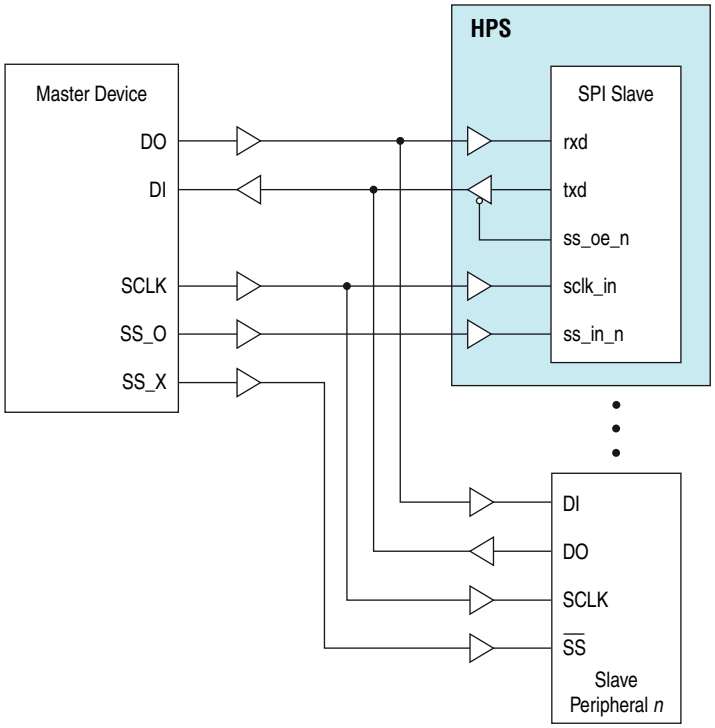
- sclk_in—serial clock to the SPI slave †
- ss_in_n—slave select input to the SPI slave †
- ss_oe_n—output enable for the SPI master or slave †
- txd—transmit data line for the SPI master or slave †
- rxd—receive data line for the SPI master or slave †

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 rxd line. The buffers shown in Figure 19–5 are external to 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 †



Slave SPI and SSP Serial Transfers †

"Motorola SPI Protocol" on page 19–13 and "Texas Instruments Synchronous Serial Protocol (SSP)" on page 19–14 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. †

Serial Transfers

"National Semiconductor Microwire Protocol" on page 19–15 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. †

Partner Connection Interfaces

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

Motorola SPI Protocol

The inactive state of the serial clock is low. The data frame can be 4 to 16 bits in length. †

Data transmission begins on the falling edge of the slave select signal. The first data bit is captured by the master and slave peripherals on the first edge of the serial clock; therefore, valid data must be present on the txd and rxd lines prior to the first serial clock edge. †

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.

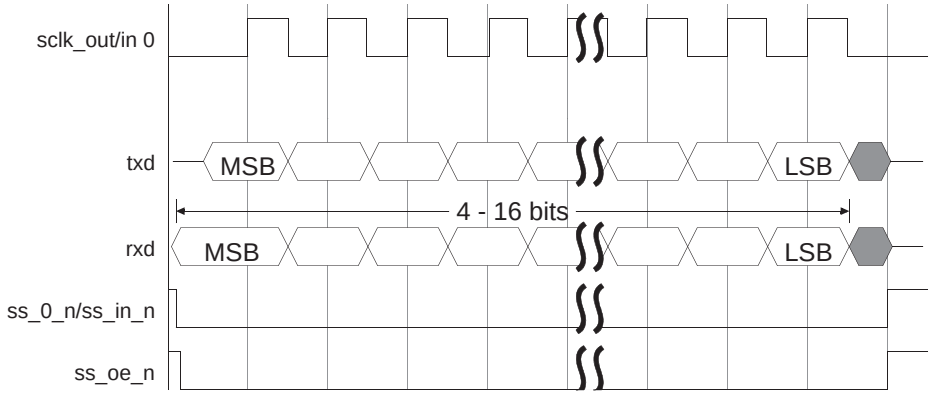
Figure 19–6 shows a timing diagram for a single SPI data transfer. †

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 †
- rxd—receive data line for the SPI master or slave †

Figure 19–6. SPI Serial Format



There are four possible transfer modes on the SPI controller for performing SPI serial transactions; refer to "Transfer Modes" on page 19–7. 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. †

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), opcode and/or EEPROM address 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.

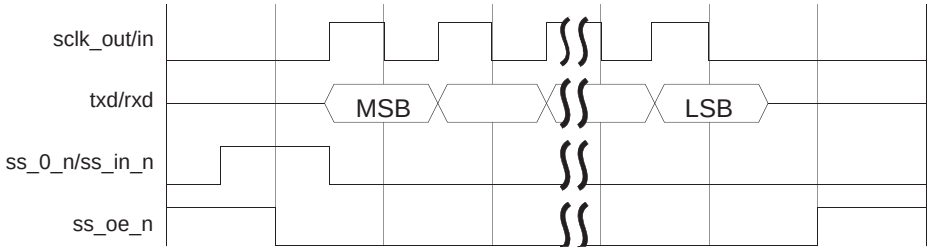
Texas Instruments Synchronous Serial Protocol (SSP)

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 rxd line. Data are propagated on the rising edge of the serial clock (sclk_out/sclk_in) and captured on the falling edge. The length of the data frame ranges from 4 to 16 bits.

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.

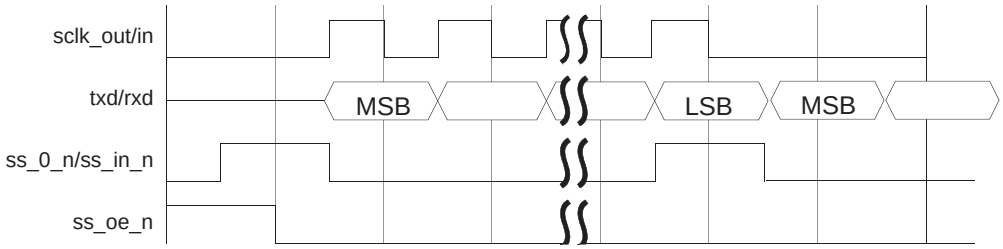
Figure 19–7 shows the timing diagram for a single SSP serial transfer. †

Figure 19–7. 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 19–8 shows the timing for a continuous SSP transfer. †

Figure 19–8. 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 `rxd` 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. †

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 (sclk_in) period later, the first bit of the control is present on the rxd line. The length of the control word can be in the range of 1 to 16 bits and is set by writing bit field CFS in the CTRLR0 register. The CFS bit field must be set to the size of the expected control word from the serial master. The remainder of the control word is received (captured on the rising edge of sclk_in) by the 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 19–9 shows the timing diagram for a single SPI serial master read from an external serial slave.

Figure 19–9. Single SPI Serial Master Microwire Serial Transfer (MDD=0)

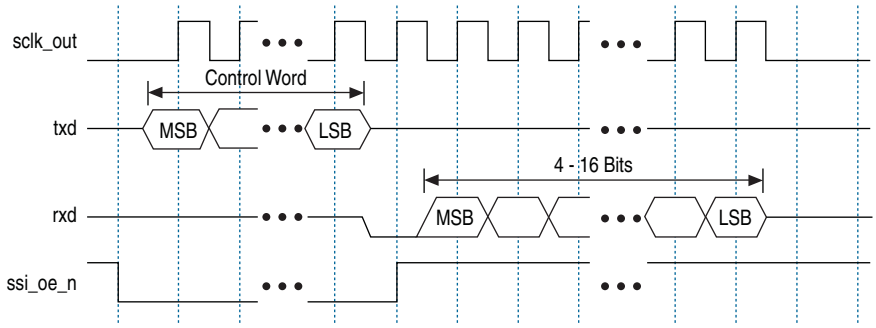
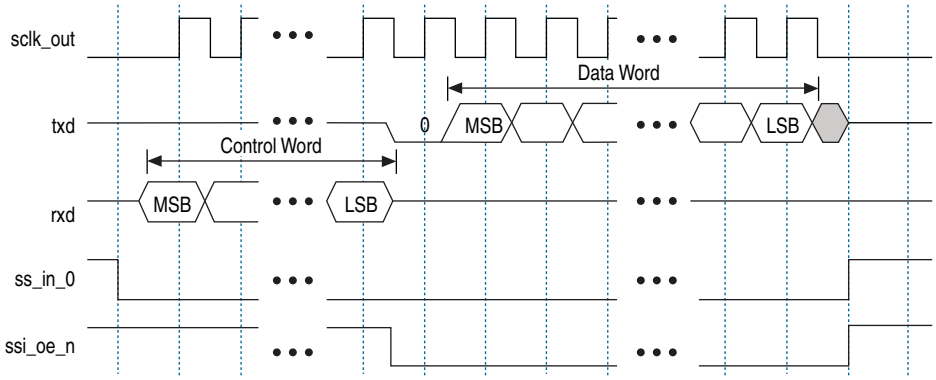


Figure 19–10 shows the timing diagram for a single SPI serial slave write to an external serial master.

Figure 19–10. 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. Accesses to the SPI peripheral are described in the following subsections. †

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. †

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 Table 19–1.

Table 19–1. 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

SPI Programming Model

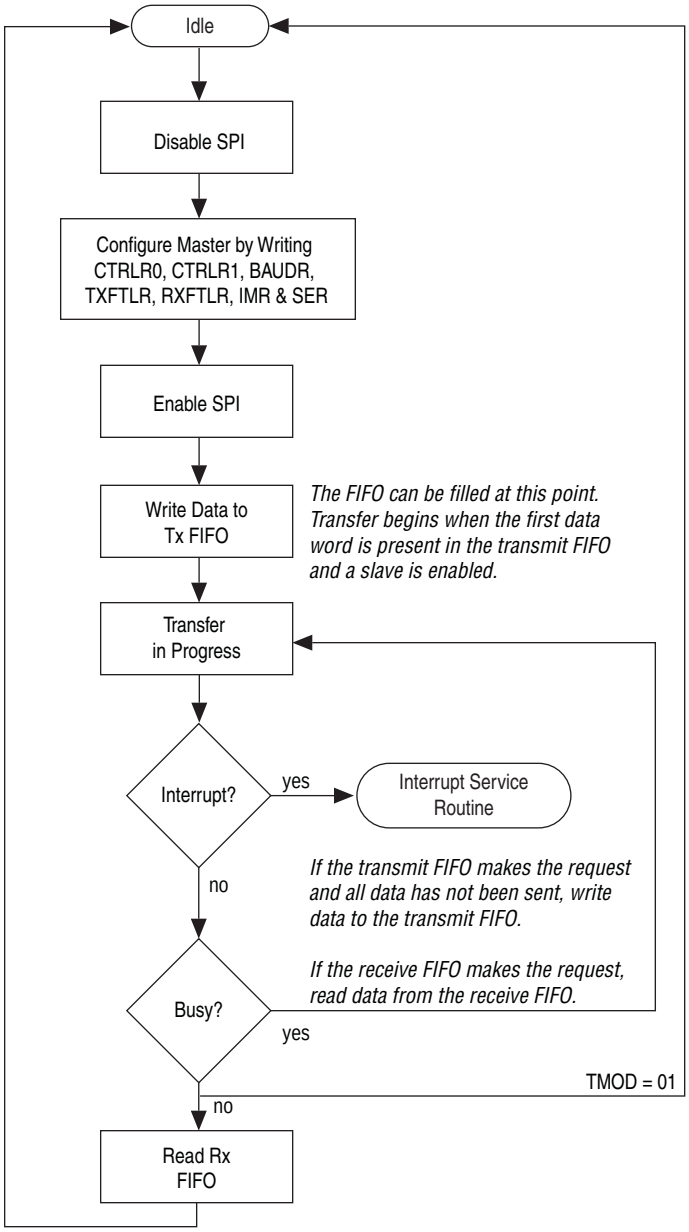
This section describes the programming model for the SPI controller based on the following master and slave transfers:

- 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 19–11 shows the software flow for a master SPI or SSP serial transfer.

Figure 19–11. Master SPI or SSP Serial Transfer Software Flow



To complete an SPI or SSP serial transfer from the SPI master, follow these steps:

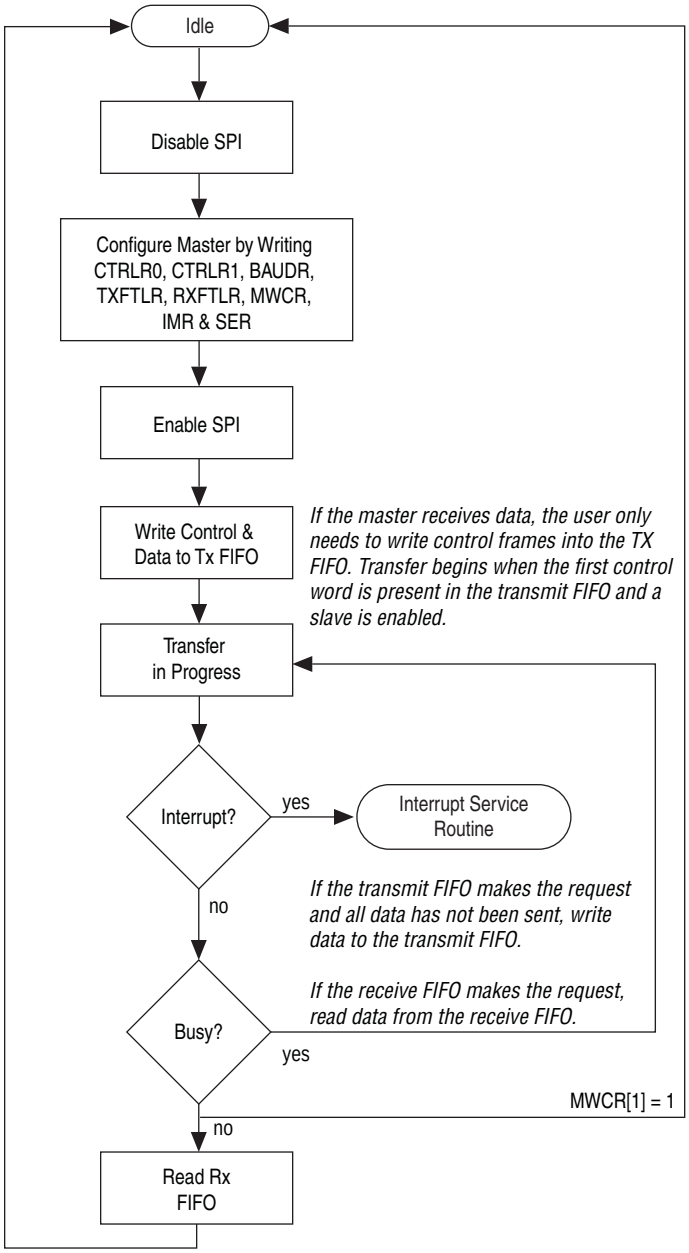
1. If the SPI master is enabled, disable it by writing 0 to the SSI Enable register (SSIENR).

2. Set up the SPI master control registers for the transfer; you can set these transfers in any order.
   - Write Control Register 0 (CTRLLR0). 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 this register with 3.
   - Write the Baud Rate Select Register (BAUDR) to set the baud rate for the transfer.
   - Write the Transmit and Receive FIFO Threshold Level registers (TXFTLR and RXFTLR) to set FIFO buffer threshold levels.
   - Write the IMR register to set up interrupt masks.
   - Write the Slave Enable Register (SER) register here to enable the target slave for selection. If a slave is enabled here, the transfer begins as soon as one valid data entry is present in the transmit FIFO 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'b10), 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 != 01), read the receive FIFO buffer until it is empty.
8. Disable the SPI master by writing 0 to SSIENR.

Master Microwire Serial Transfers

Figure 19–12 shows the software flow for a Microwire serial transfer.

Figure 19–12. Microwire Serial Transfer Software Flow



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.

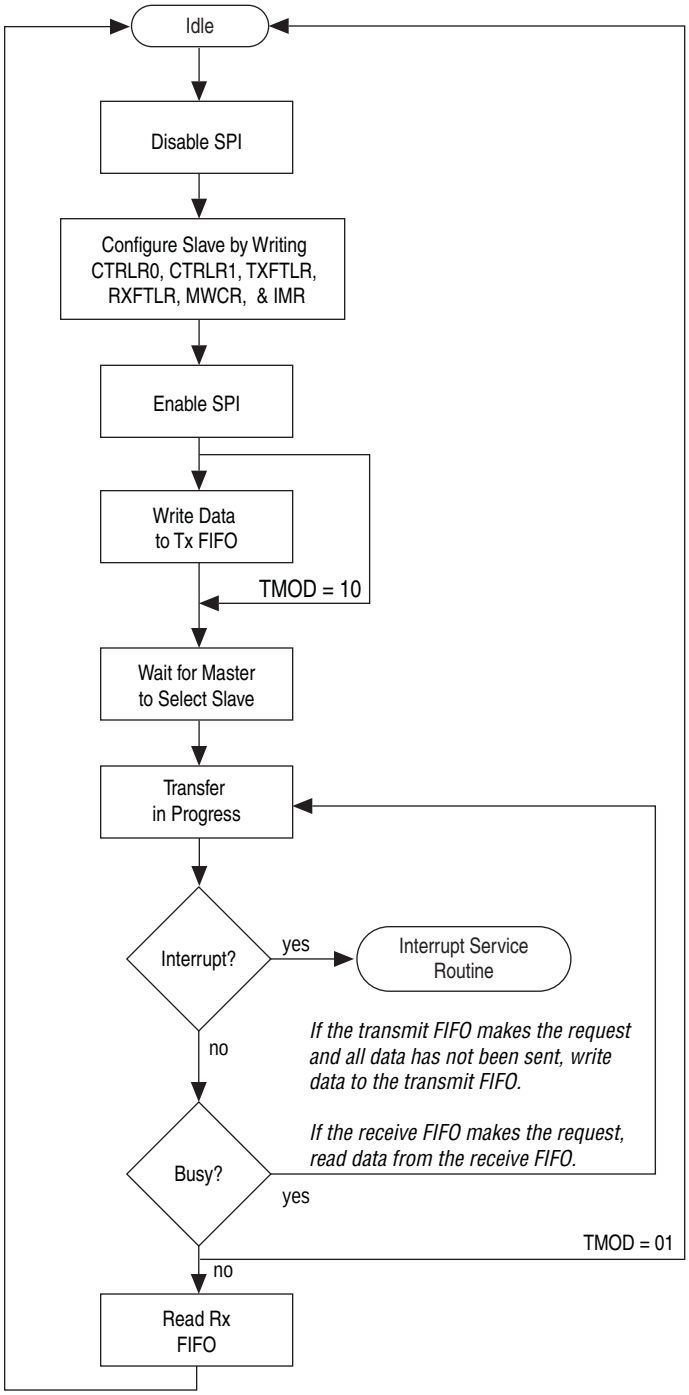
2. Set up the SPI control registers for the transfer. You can set these registers in any order.
   - Write CTRLR0 to set transfer parameters. If the transfer is sequential and the SPI master receives data, write CTRLR1 with the number of frames in the transfer minus 1. For example, if you want to receive four data frames, write this register with 3.
   - 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.

Slave SPI and SSP Serial Transfers

Figure 19–13 shows the software flow for a slave SPI or SSP serial transfer.

Figure 19–13. 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=2'b00) or transmit only (TMOD=2'b01), write data for transmission to the master into the transmit FIFO buffer (write DR). If the transfer mode is receive only (TMOD=2'b10), 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 != 01), read the receive FIFO buffer until empty.
9. Disable the SPI slave by writing 0 to SSIENR.

Slave Microwire Serial Transfers

For SPI slave, 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.

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. The following examples show the software flow for slave selection:

- 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 logic '1' to select slave 1 in this example.

8. Write SSIENR register bit[0] to logic '1' to enable SPI master.

- For SPI slave:

9. If the SPI slave is enabled, disable it by writing 0 to SSIENR.
10. Write CTRLR0 to match the required transfer.
11. Write TXFTLR and RXFTLR to set FIFO buffer threshold levels.
12. Write IMR register to set interrupt masks.
13. Write SSIENR register bit[0] to logic '1' to enable SPI slave.
14. If the SPI slave transmits data, write data into TX FIFO buffer. Note all other SPI slaves are disabled (SSIENR = 0) and therefore will 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.†

For details about the DMA controller, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

DMA Operation

For more information about the DMA operation, refer to the ARM DMA 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.†

For details about the DMA burst length microcode setup, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 will run out of data (underflow). To prevent this condition, you must set the watermark level correctly.†

Transmit Watermark Level

Consider the example where the assumption is made: †

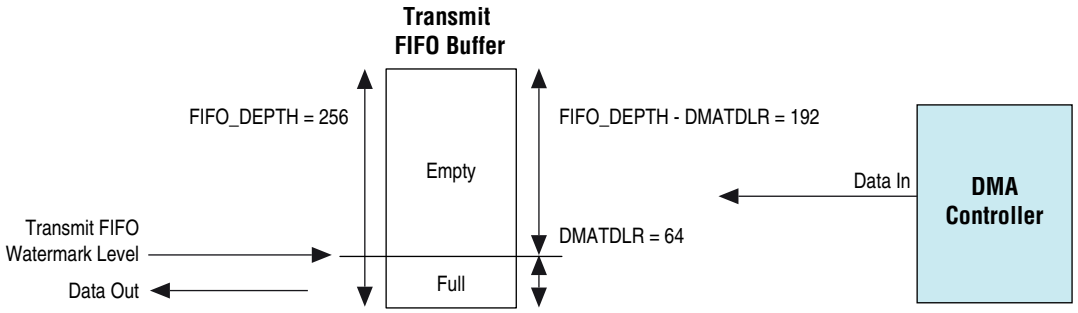
DMA burst length = FIFO_DEPTH - DMATDLR

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 two different watermark level settings. †

- Case 1: DMATDLR = 64: †
  - Transmit FIFO watermark level = DMATDLR = 64: †
  - DMA burst length = FIFO_DEPTH - DMATDLR = 192: †
  - SPI transmit FIFO_DEPTH = 256: †
  - Block transaction size = 960: †

Figure 19–14 shows the transmit FIFO buffer when the watermark level equals 64.

Figure 19–14. Transmit FIFO Watermark Level = 64



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

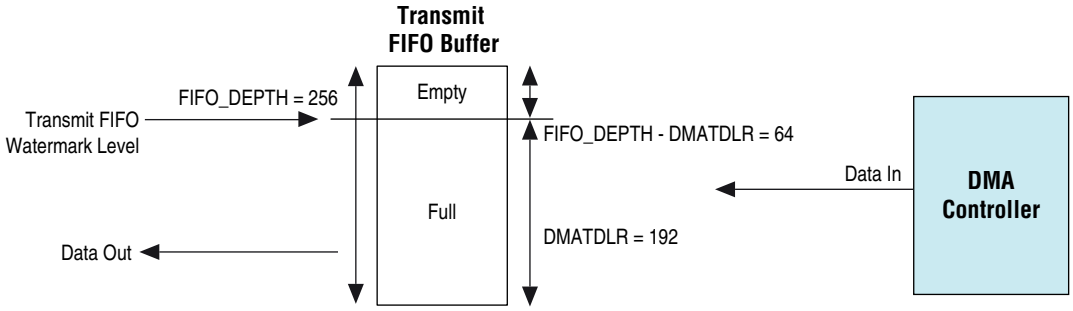
Block transaction size/DMA burst length = 960/192 = 5

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

- Case 2: DMATDLR = 192†
  - Transmit FIFO watermark level = DMATDLR = 192 †
  - DMA burst length = FIFO_DEPTH - DMATDLR = 64 †
  - SPI transmit FIFO_DEPTH = 256 †
  - Block transaction size = 960 †

Figure 19–15 shows the transmit FIFO buffer when the watermark level equals 192.

Figure 19–15. 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. †

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 SPI transmits data to the rate at which the DMA can respond to destination burst requests. †

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: †

DMA burst length <= FIFO_DEPTH - DMATDLR

In case 2: DMATDLR = 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: †

DMA burst length = FIFO_DEPTH - 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 will 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. †

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 DMATDLR + 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 will fill 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, DMATDLR + 1, should be set to minimize the probability of overflow, as shown in Figure 8. It is a trade off between the number of DMA burst transactions required per block versus the probability of an overflow occurring. †

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: †

DMA burst length = DMATDLR + 1

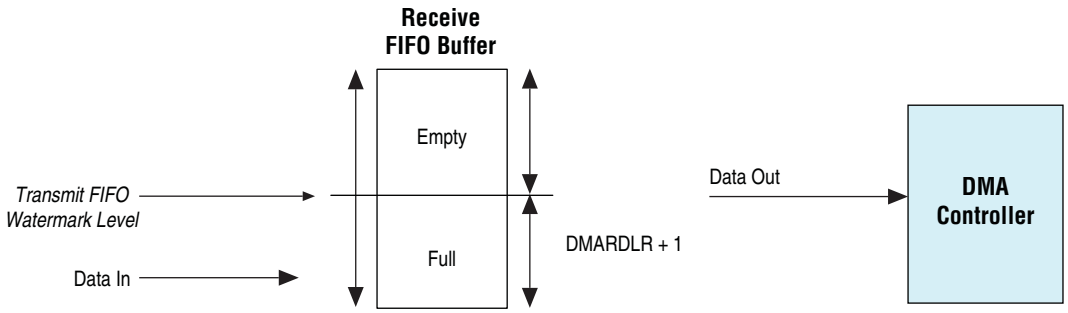
If the number of data items in the receive FIFO buffer is equal to the source burst length at the time of the burst request is made, the receive FIFO buffer may be emptied, but not underflowed, at the completion of the burst transaction. For optimal operation, DMA burst length should be set at the watermark level, DMATDLR + 1. †

Adhering to this equation reduces the number of DMA bursts in a block transfer, which in turn can improve bus utilization. †

The receive FIFO buffer will 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 19–16 shows the receive FIFO buffer.

Figure 19–16. Receive FIFO Buffer



SPI Controller Address Map and Register Definitions

The address map and register definitions reside in hps.html file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for any of the following module instances:

- **spis0**
- **spis1**
- **spim0**
- **spim1**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 19–2 shows the revision history for this document.

Table 19–2. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	Added programming model, address map and register definitions, clocks, and reset sections.
January 2012	1.0	Initial release.

20. I²C Controller

cv_54020-1.2

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 kilobits per second (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. †

Features of the I²C Controller

The I²C controller has the following features:

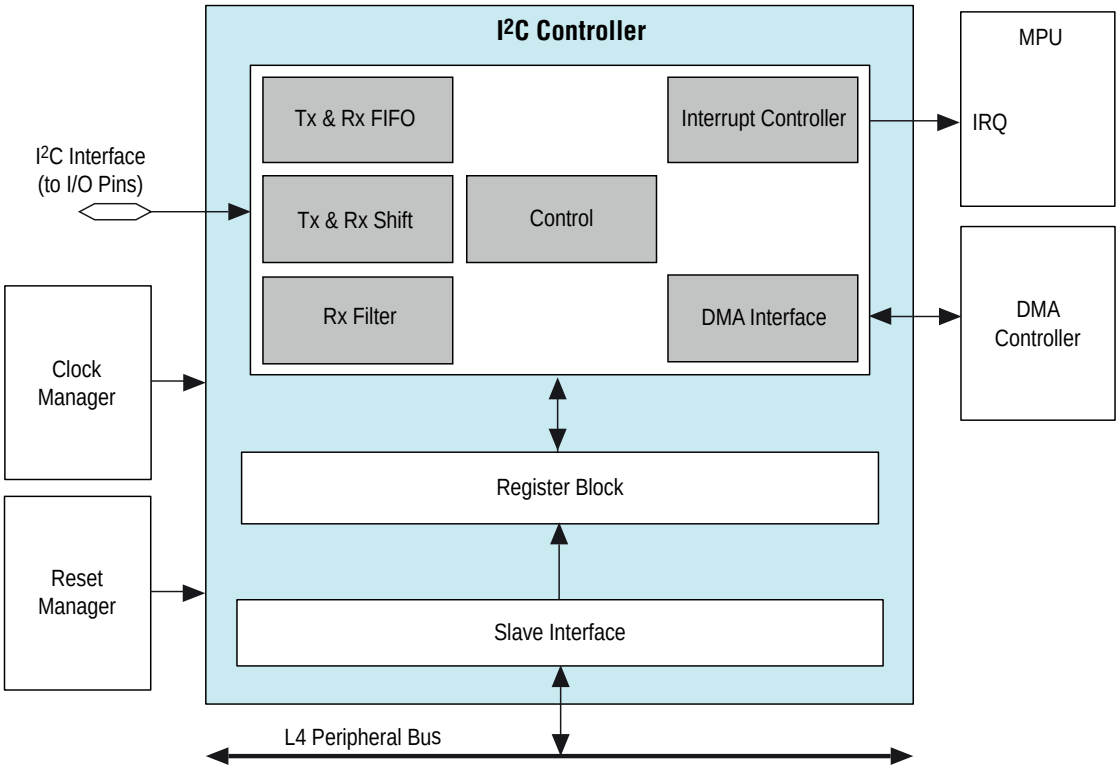
- Maximum clock speed of up to 400 Kbps
- One of the following I²C operations:
  - A master in an I²C system and programmed only as a master †
  - A slave in an I²C system and programmed only as a slave †
- 7- or 10-bit addressing †
- Mixed read and write combined-format transactions in both 7-bit and 10-bit addressing mode †
- Bulk transmit mode †
- Transmit and receive buffers †
- Handles bit and byte waiting at all bus speeds †
- DMA handshaking interface †

I2C Controller Block Diagram and System Integration

The I2C controller consists of a 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 shows the I2C controller block diagram.

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 interrupt 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. †

Functional Description of the I²C Controller

This section describes the functional operation of the I²C controller.

Feature Usage

The I²C controller can operate in standard mode (with data rates 0 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 either mode:

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

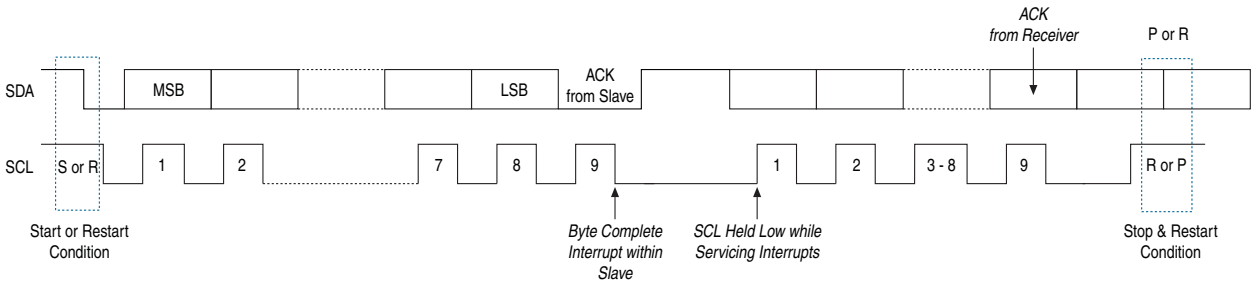
The master is responsible for generating the clock and controlling the transfer of data. The slave is responsible for either transmitting or receiving data to/from the master. The acknowledgement of data is sent by the device that is receiving data, which can be either a master or a slave. As mentioned 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 20–2 illustrates the data transfer behavior on the I2C bus.

Figure 20–2. 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. Allowing the transmit FIFO to empty causes the I²C controller to generate a STOP condition on the I2C bus. †

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 `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 transmit FIFO is empty when the current I²C transfer completes, a STOP is issued and the next transfer is issued following a START condition. †

Protocol Details

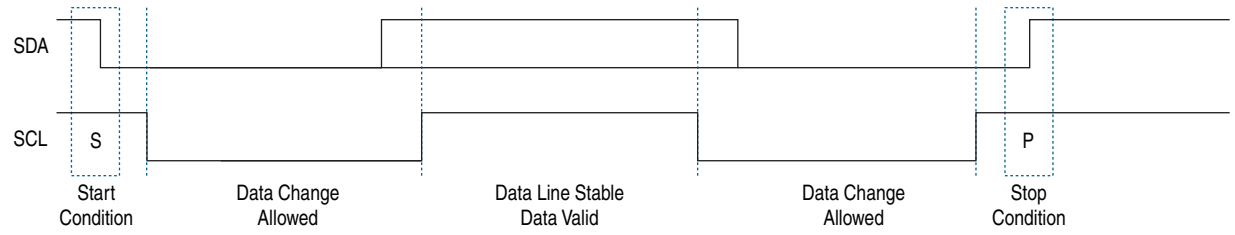
This section describes the I²C controller protocols.

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. †

Figure 20–3 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 20–3. Timing Diagram for START and STOP Condition †



The signal transitions for the START or STOP condition, as shown in Figure 20–3, 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. †

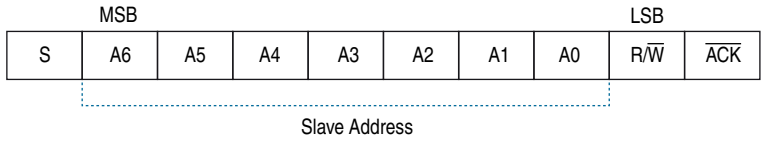
Addressing Slave Protocol

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

7-Bit Address Format

During the 7-bit address format, the first seven bits (bits 7:1) of the first byte set the slave address and the LSB bit (bit 0) is the R/W bit as shown in Figure 20–4. 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 20–4. 7-bit Address Format †

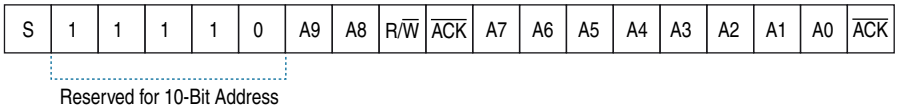


S: Start Condition
R/W: Read/Write Pulse
ACK: Acknowledge (sent by slave)

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 20–5 shows the 10-bit address format. †

Figure 20–5. 10-Bit Address Format †



S: Start Condition
R/W: Read/Write Pulse
ACK: Acknowledge (sent by slave)

Table 20–1 defines the special purpose and reserved first byte addresses. †

Table 20–1. I²C Definition of Bits in First Byte †

Slave Address	R/W Bit	Description
0000 000	0	General call address. The I²C 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" on page 20–9
0000 001	X	CBUS address. The I²C 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 20–1:
(1) 'X' indicates do not care.

Transmitting and Receiving Protocol

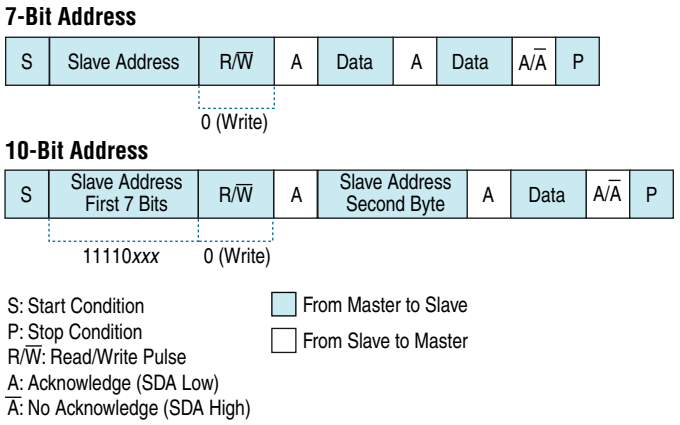
The master can initiate data transmission and reception to or from the bus, acting as either a master-transmitter or master-receiver. A slave responds to requests from the master to either transmit data or receive data to or from the bus, acting as either a slave-transmitter or slave-receiver, respectively.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

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 Figure 20–6, then the slave-receiver responds to the master-transmitter with an ACK pulse after every byte of data is received. †

Figure 20–6. Master-Transmitter Protocol †



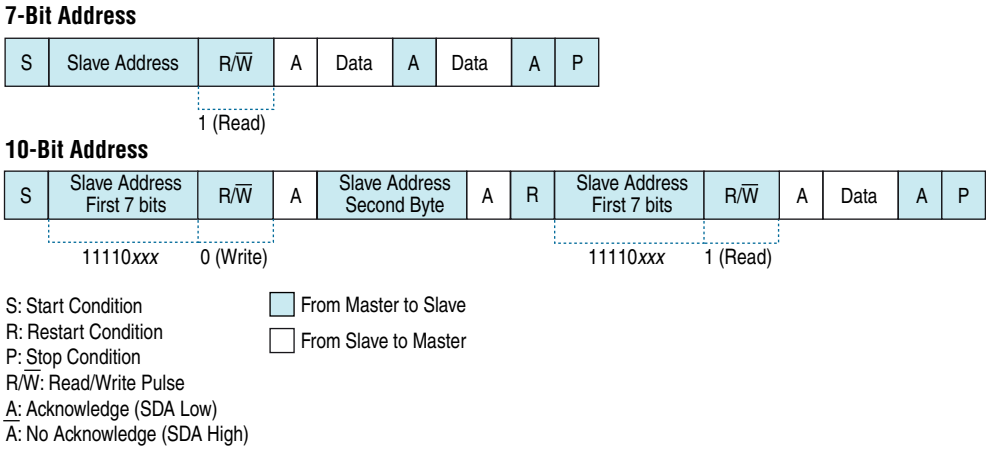
Master-Receiver and Slave-Transmitter

If the master is receiving data as shown in Figure 20–7, 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" on page 20–5. †

The I²C controller must be inactive on the serial port (I2C_DYNAMIC_TAR_UPDATE = 1) before the target slave address register, IC_TAR can be reprogrammed. †

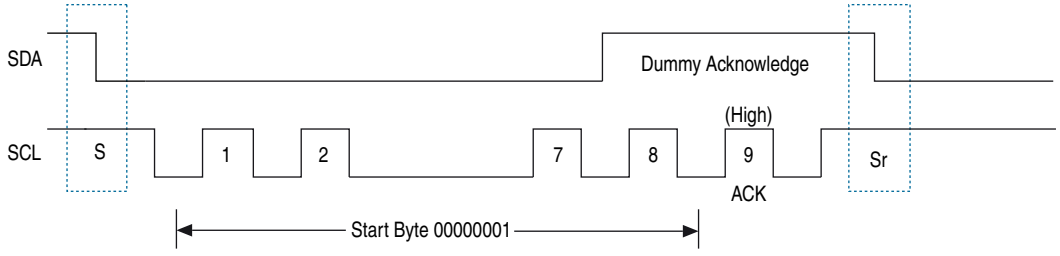
Figure 20–7. Master-Receiver Protocol †



START BYTE Transfer Protocol

The START BYTE transfer protocol is set up for systems that do not have an on-board dedicated I²C hardware module. When the 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 Figure 20–8. 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 20–8. 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

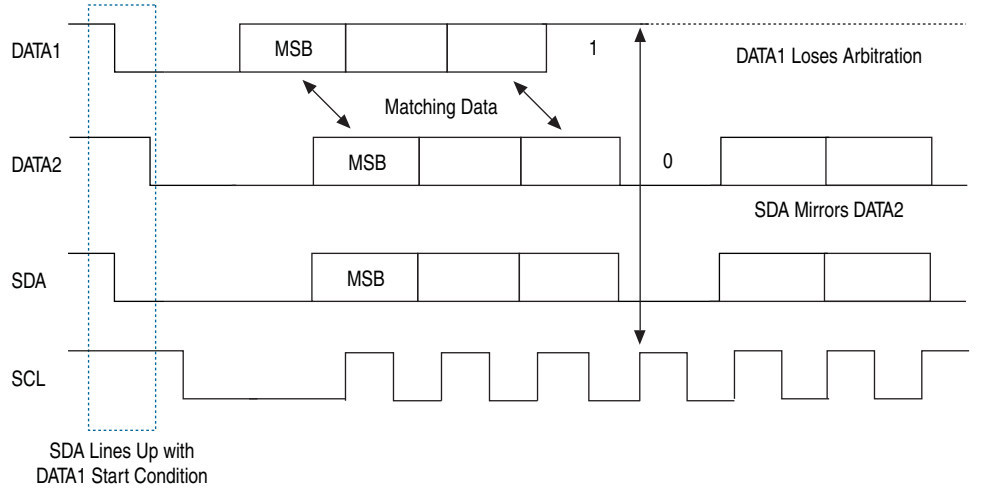
The I2C controller bus protocol allows multiple masters to reside on the same bus. If there are two masters on the same I2C-bus, there is an arbitration procedure if both try to take control of the bus at the same time by simultaneously generating a START condition. Once a master (for example, a microcontroller)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

Figure 20–9 illustrates the timing of when two masters are arbitrating on the bus.

Figure 20–9. 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 †

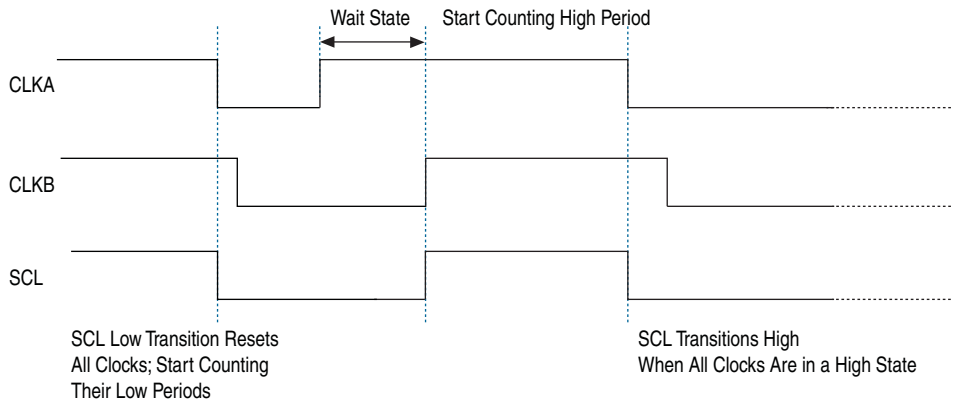
Slaves are not involved in the arbitration process. †

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 Figure 20–10. Optionally, slaves may hold the SCL line low to slow down the timing on the I²C bus. †

Figure 20–10. Multiple Master Clock Synchronization †



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 `l4_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 `l4_sp_clk` cycles incurs. †

Consequently, the minimum SCL low time of which the I2C controller is capable is nine (9) `l4_sp_clk` periods (8+1), while the minimum SCL high time is thirteen (13) `l4_sp_clk` periods (6+1+3+3). †

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 l4_sp_clk clock pulses required for setting the proper SCL clocks high and low times is as follows: †

Equation 20–1.

```
IC_HCNT = ceil(MIN_SCL_HIGHtime*OSCFREQ)
IC_LCNT = ceil(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 Mbs, bus loading = 100pF
160 ns for 3.4 Mbs, bus loading = 400pF

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

OSCFREQ = l4_sp_clk clock frequency (Hz)
```

For example:

```
OSCFREQ = 100 MHz
I2Cmode = fast, 400 kbps
MIN_SCL_HIGHtime = 600 ns
MIN_SCL_LOWtime = 1300 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 †
```

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. †

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. †

For details about the DMA controller, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

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.

For more information, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Resets

Each I²C controller has a separate reset signal. The reset manager drives the signals on a cold or warm reset.

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Interface Pins

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 20–2 shows I/O pin use of the I²C controller interface.

Table 20–2. I²C Controller Interface Pins

Pin Name	Signal Width	Direction	Description
SCL	1 bit	Bidirectional	Serial clock
SDA	1 bit	Bidirectional	Serial data

I²C Controller Programming Model

This section describes the programming model for the I²C controllers based on the two master and slave operation modes. †

☞ Each I²C controller should be set to operate only as an I²C master or as an I²C slave, never 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

This section discusses slave mode procedures. †

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. †

   ☞ 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). †

   ☞ 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. †

   a. 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. †

   b. Software must then act to satisfy the I2C transfer. †

   c. The timing interval used should be in the order of 10 times the fastest SCL clock period the I2C controller can handle. For example, for 400 Kbps, the timing interval is 25 us. †

   ☞ 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. †

   ☞ 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 C_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. †

   a. Reads that indicate bit 6 (R_TX_ABRT) being set to 1 must be treated as the equivalent of the TX_ABRT interrupt being asserted. †

   b. There is no further action required from software. †

   c. The timing interval used should be similar to that described in the previous step for the IC_RAW_INTR_STAT[5] register. †

5. Software writes to the 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 and/or TX_ABRT interrupts have been masked, then clearing of the IC_RAW_INTR_STAT register will have already been performed when either the R_RD_REQ or R_TX_ABRT bit has been read as 1. †

7. The 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. †

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

Master Mode Operation

This section discusses master mode procedures. †

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: †

1. Disable the I2C controller by writing 0 to the IC_ENABLE register. †
2. Write to the IC_CON register to set the maximum speed mode supported for slave operation (bits 2:1) and to specify whether the 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 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. †

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

Dynamic IC_TAR or IC_10BITADDR_MASTER Update

The 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. The I²C controller in master mode continues to initiate transfers as long as there are commands present in the transmit FIFO. If the transmit FIFO becomes empty, the I²C controller inserts a STOP condition after completing the current transfers. †

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.†

For details about the DMA controller, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

The FIFO buffer depth (FIFO_DEPTH) for both the RX and TX buffers in the I²C controller is 64 entries.

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. †

For details about the DMA burst length microcode setup, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

Data should be fetched from the DMA often enough for the transmit FIFO to perform serial transfers continuously, that is, when the FIFO begins to empty, another DMA request should be triggered. Otherwise, the FIFO will run out of data (underflow) causing a STOP to be inserted on the I²C bus. To prevent this condition, you must set the watermark level correctly. †

Transmit Watermark Level

Consider the example where the assumption is made: †

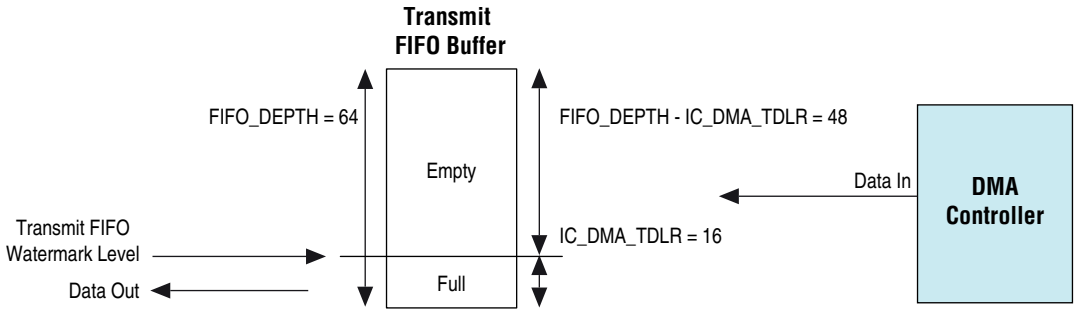
DMA burst length = FIFO_DEPTH - IC_DMA_TDLR †

Here the number of data items to be transferred in a DMA burst is equal to the empty space in the transmit FIFO. Consider 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 20–11 shows the transmit FIFO when the watermark level equals 16.

Figure 20–11. Transmit FIFO Watermark Level = 16



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

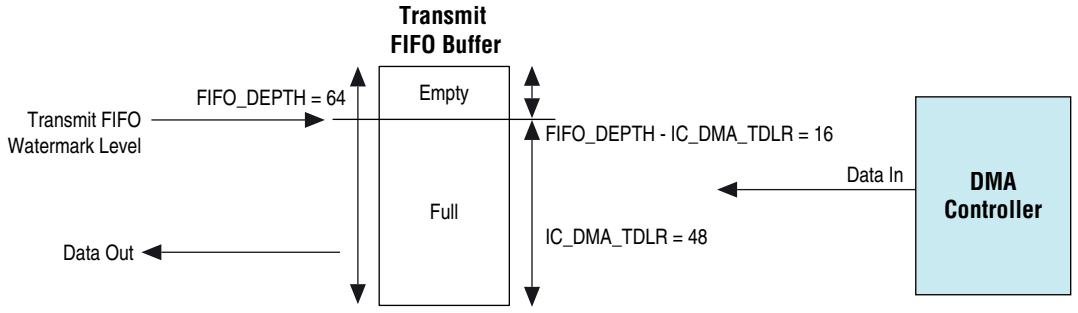
Block transaction size/DMA burst length = 240/48 = 5

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 20–12 shows the transmit FIFO when the watermark level equals 48.

Figure 20–12. 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: †

DMA burst length <= FIFO_DEPTH - 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: †

DMA burst length = FIFO_DEPTH - 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 will 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 will fill 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 Figure 20–13. 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: †

DMA burst length = IC_DMA_RDLR + 1

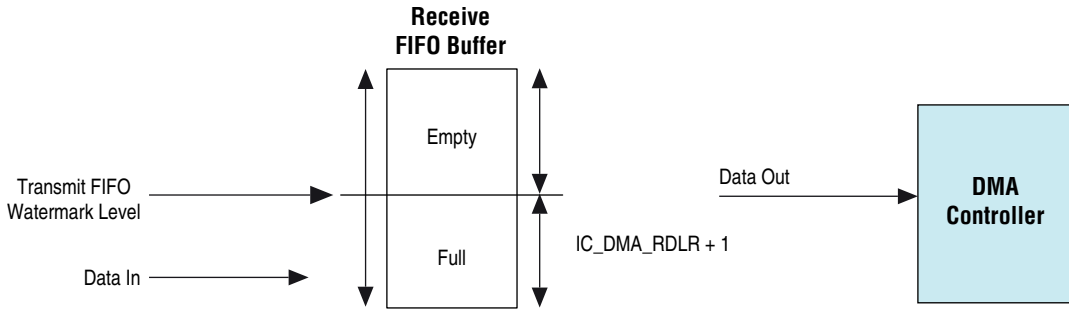
If the number of data items in the receive FIFO is equal to the source burst length at the time of the burst request is made, the receive FIFO may be emptied, but not underflowed, at the completion of the burst transaction. For optimal operation, DMA burst length should be set at the watermark level,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

The receive FIFO will 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 20–13 shows the receive FIFO buffer.

Figure 20–13. Receive FIFO Buffer



I²C Controller Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for any of the following module instances:

- **i2c0**
- **i2c1**

- **i2c2**
- **i2c3**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 20–3 shows the revision history for this document.

Table 20–3. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	Added programming model, address map and register definitions, clocks, reset, and interface pins sections.
January 2012	1.0	Initial release.

21. UART Controller

cv_54021-1.2

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.

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-bit transmit and receive FIFO buffer depth
  - 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 †

UART Controller Block Diagram and System Integration

Figure 21–1 shows the integration of the UART controller in the HPS system.

Figure 21–1. UART Block Diagram

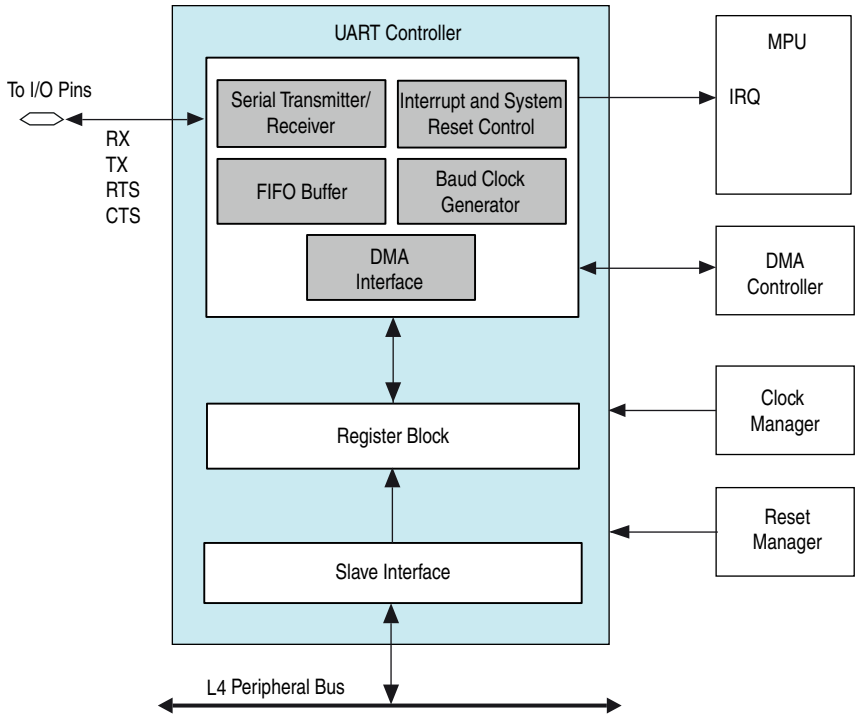


Table 21–1 provides a summary description of the major blocks in the UART controller.

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. †

Table 21–1. UART Controller Block Descriptions

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. For more specific information, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone® V Device Handbook*.

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. †

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 (IIR_FCR.FIFOE bit is set)

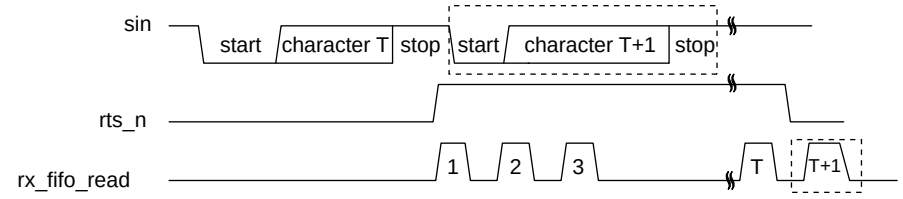
When automatic RTS is enabled, the rts_n output pin is forced inactive (high) when the receive FIFO buffer level reaches the threshold set by RCVR trigger (IIR_FCR.RT). When rts_n is connected to the cts_n input pin of another UART device, the other UART stops sending serial data until the receive FIFO buffer has available space (until it is completely empty). †

The selectable receive FIFO buffer threshold values are 1, ¼, ½, 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), signalling 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. Figure 21–2 illustrates automatic RTS operation. In Figure 21–2, the character T is received because rts_n is not detected prior to the next character entering the sending UART transmitter. †

Figure 21–2. 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. †

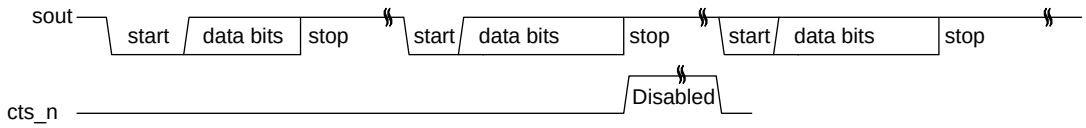
If the cts_n input is not deactivated before the middle of the last stop bit, another character is transmitted before the transmitter is disabled. While the transmitter is disabled, you can continue to write and even overflow to the transmit FIFO buffer. †

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 IIR_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 21–3 timing diagram shows auto CTS operation.†

Figure 21–3. Automatic CTS Timing



Clocks

The UART controller is connected to the l4_sp_clk clock. The clock input is driven by the clock manager.

For more information about the clock manager, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

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.

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Interrupts

The assertion of the UART interrupt output signal occurs when one of the following interrupt types are enabled and active: †

Table 21–2. Interrupt Types and Priority †

Interrupt Type	Priority
Receiver line status	Highest
Received data available	Second

Table 21–2. Interrupt Types and Priority †

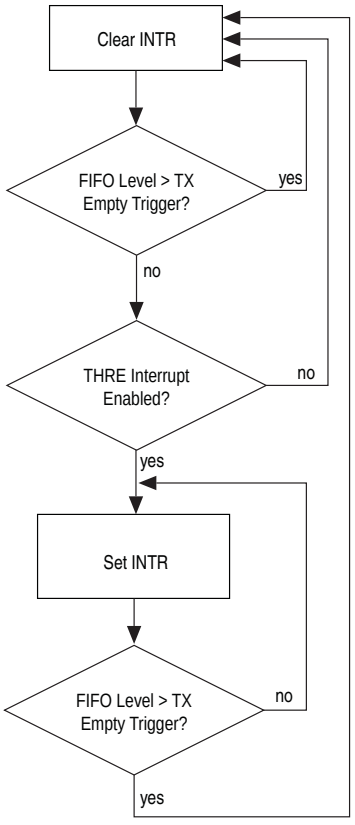
Interrupt Type	Priority
Character timeout indication	Second
Transmit holding register empty	Third

You can enable the interrupt types with the interrupt enable register (IER_DLH).

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 in Figure 21–4. †

Figure 21–4. Programmable THRE Interrupt Mode

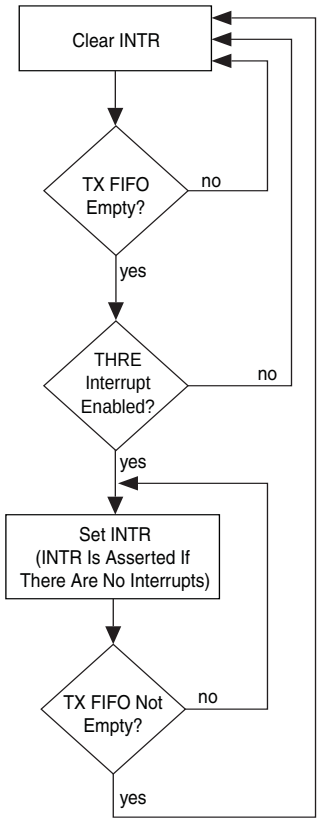


The threshold level is programmed into IIR_FCR.TET. The available empty thresholds are empty, 2, ¼, and ½. The optimum threshold value depends on the system's ability to begin a new transmission sequence in a timely manner. However, one of these thresholds should prove optimum in increasing system performance 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 THR or FIFO buffer. Figure 21–5 illustrates THRE interrupt generation when not in programmable THRE interrupt mode.

Figure 21–5. Interrupt Generation without Programmable THRE Interrupt Mode



UART Controller Programming Model

This section describes the programming model for the UART controller.

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.

For details about the DMA controller, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

The FIFO buffer depth (FIFO_DEPTH) for both the RX and TX buffers in the UART controller is 128 entries.

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 (IIR_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. †

For details about the DMA burst length microcode setup, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

Data should be fetched from the DMA often enough for the transmit FIFO to perform serial transfers continuously, that is, when the FIFO begins to empty, another DMA request should be triggered. Otherwise, the FIFO will run out of data (underflow) causing a STOP to be inserted on the UART bus. To prevent this condition, you must set the watermark level correctly. †

Transmit Watermark Level

Consider the example where the following assumption is made: †

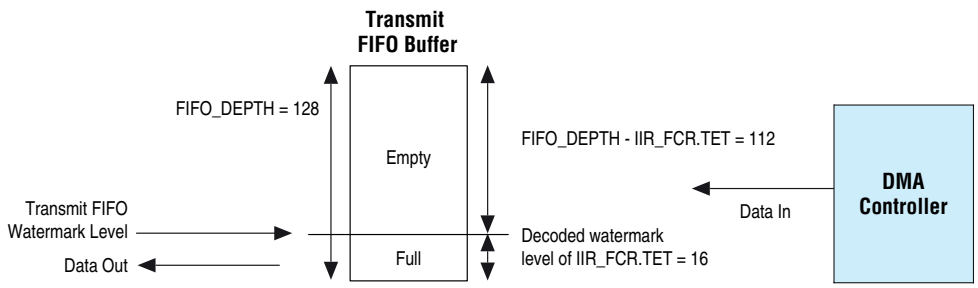
DMA burst length = FIFO_DEPTH - decoded watermark level of IIR_FCR.TET †

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: †

- Case 1: IIR_FCR.TET = 1, which 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 21–6 shows the transmit FIFO when the watermark level equals 16.

Figure 21–6. Transmit FIFO Watermark Level = 16



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

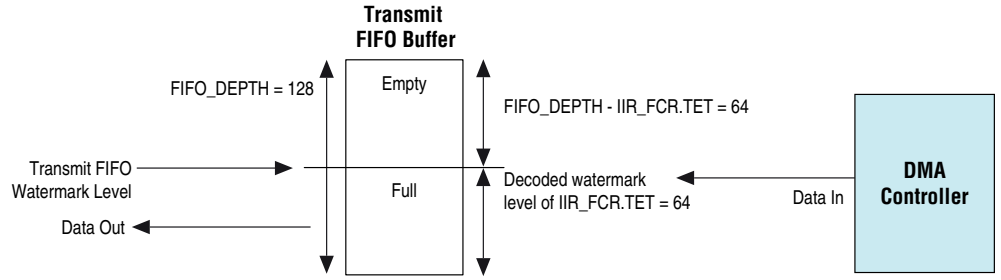
Block transaction size/DMA burst length = 448/112 = 4

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.

- Case 2: `IIR_FCR.TET` = 3, which 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 21–7 shows the transmit FIFO when the watermark level equals 64.

Figure 21–7. 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 <= 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 should be set at the FIFO level that triggers a transmit DMA request; that is: †

DMA burst length = FIFO_DEPTH - decoded watermark level of 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 will 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 will fill 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 Figure 21–8. 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: †

DMA burst length = decoded watermark level of IIR_FCR.RT + 1

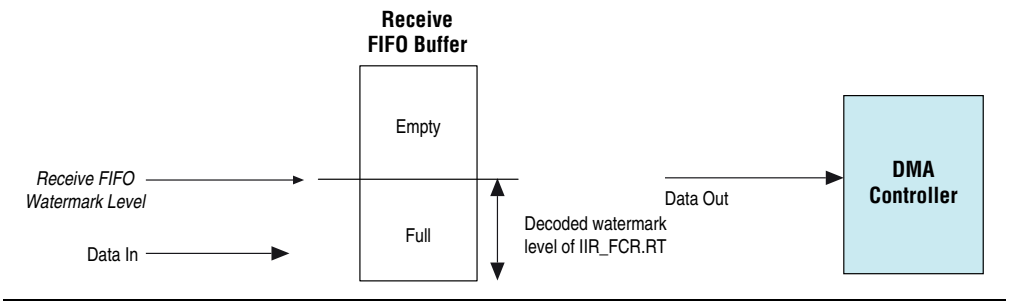
If the number of data items in the receive FIFO is equal to the source burst length at the time of the burst request is made, the receive FIFO may be emptied, but not underflowed, at the completion of the burst transaction. For optimal operation, DMA burst length should be set at the watermark level, 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 will 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 21–8 shows the receive FIFO buffer.

Figure 21–8. Receive FIFO Buffer



UART Controller Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for either of the following module instances:

- **uart0**
- **uart1**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 21–3 shows the revision history for this document.

Table 21–3. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	Added programming model, address map and register definitions, and reset sections.
January 2012	1.0	Initial release.

22. General-Purpose I/O Interface

cv_54022-1.2

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.

Features of the GPIO Interface

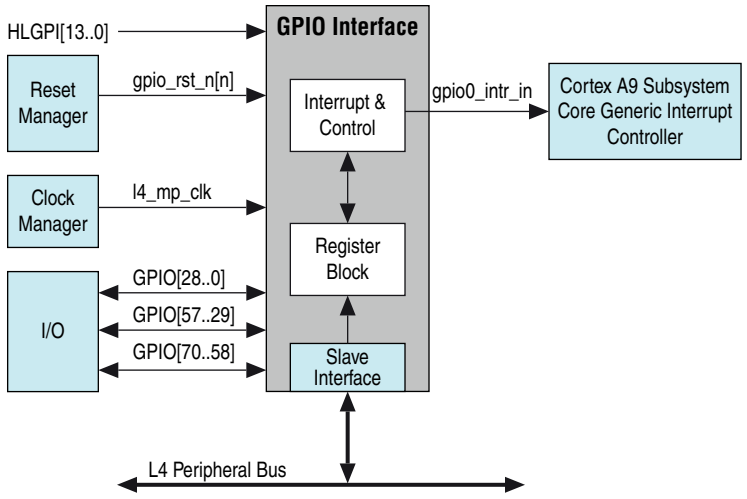
The GPIO interface offers the following features:

- Supports digital de-bounce
- Configurable interrupt mode
- Supports up to 71 I/O pins and 14 input-only pins

GPIO Interface Block Diagram and System Integration

Figure 22–1 shows a block diagram of the GPIO interface.

Figure 22–1. GPIO Interface Block Diagram



Functional Description of the GPIO Interface

This section provides functional details of the GPIO interface.

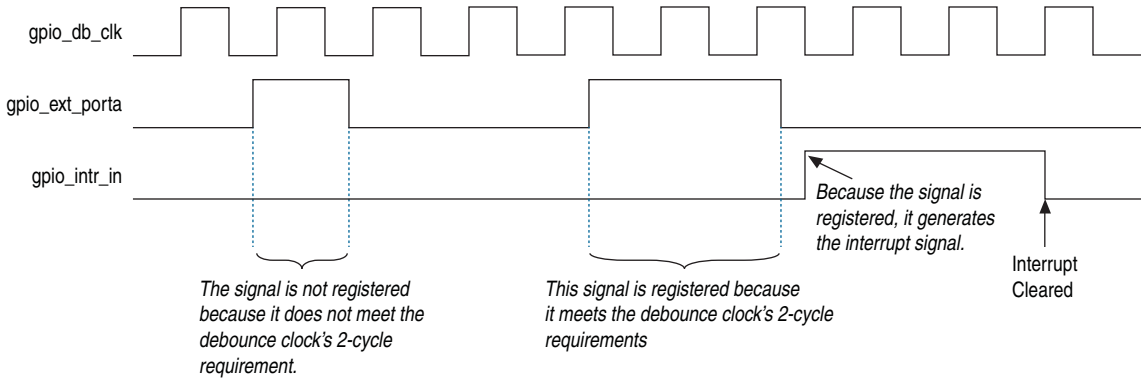
Debounce Operation

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 interrupt 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.

Figure 22–2 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



Enabling the debounce circuitry increases interrupt latency by two clock cycles of the debounce clock.

Pin Directions

The pins `GPIO0` through `GPIO70` can be configured to be either input or output signals. The pins `HLGPI0` through `HLGPI13` share pins with the HPS DDR controller and are input-only signals.

GPIO Interface Programming Model

Debounce capability for each of the input signals on port A 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.

GPIO Interface Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for any of the following module instances:

- **gpio0**
- **gpio1**
- **gpio2**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 22–1 shows the revision history for this document.

Table 22–1. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	Added programming model section.
January 2012	1.0	Initial release.

23. Timer

cv_54023-1.2

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.

The microprocessor unit (MPU) subsystem provides additional timers. For information about the timers in the MPU, refer to the *Cortex-A9 MPU System* chapter in volume 3 of the *Cyclone V Device Handbook*.

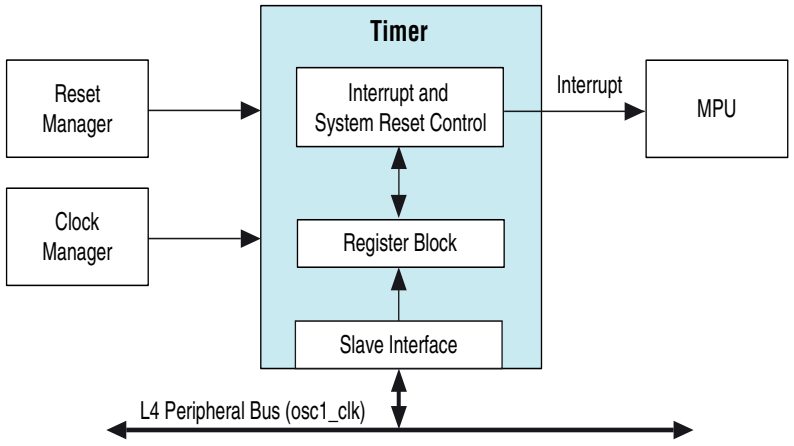
Features of the Timer

- Supports interrupt generation
- Supports free-running mode
- Supports user-defined count mode

Timer Block Diagram and System Integration

Figure 23–1 shows a block diagram of the timer. 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.

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 shows the clock signals and connections associated with the timers.

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		

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.

For more information about clock performance, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Resets

The timers are reset by a cold or warm reset. Resetting the timers produces the following results:

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 returns the value 0 and 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.

Timer Programming Model

This section describes the programming model for the timer.

Initialization

To initialize the timer, perform the following steps: †

1. Initialize the timer through the `timer1controlreg` register: †
   a. Disable the timer by writing a 0 to the timer1 enable bit (`timer1_enable`) of the `timer1controlreg` register. †

   ☞ Before writing to a timer1 load count 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. †

   b. 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. †
   c. 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 or Disabling the Timer

To enable the timer, write a 1 to the `timer1_enable` bit of the `timer1controlreg` register. To disable the timer, write a 0 to the `timer1_enable` bit. †

When a timer transitions to the enabled state, the current value of `timer1loadcount` register is loaded into the timer counter. †

When the timer enable bit is cleared to 0, the timer counter and any associated registers in the timer clock domain, are asynchronously reset. †

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

This section discusses various interrupt cases and how to service them.

Clearing Interrupt

You can clear an active timer interrupt by reading the `timer1eoi` register or by disabling the timer. When the timer is enabled, its interrupt remains asserted until it is cleared by reading the `timer1eoi` register. †

If you clear the interrupt at the same time as the timer reaches 0, the interrupt remains asserted. This is because setting the timer interrupt takes precedence over clearing the interrupt. †

Checking Interrupt Status

You can query the interrupt status of the timer without clearing its interrupt. To check the interrupt status, read the `timer1intstat` register. †

Masking 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 reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for any of the following module instances:

- **osc1timer0**
- **osc1timer1**
- **sptimer0**
- **sptimer1**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 23–2 shows the revision history for this document.

Table 23–2. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	Added programming model and address map and register definitions sections.
January 2012	1.0	Initial release.

24. Watchdog Timer

cv_54024-1.2

The primary function of a watchdog timer is to provide a way for the system to recover from an unresponsive state. 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.

☞ The microprocessor unit (MPU) subsystem provides two additional watchdog timers. For information about the watchdog timers in the MPU, refer to the *Cortex-A9 MPU System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Features of the Watchdog Timer

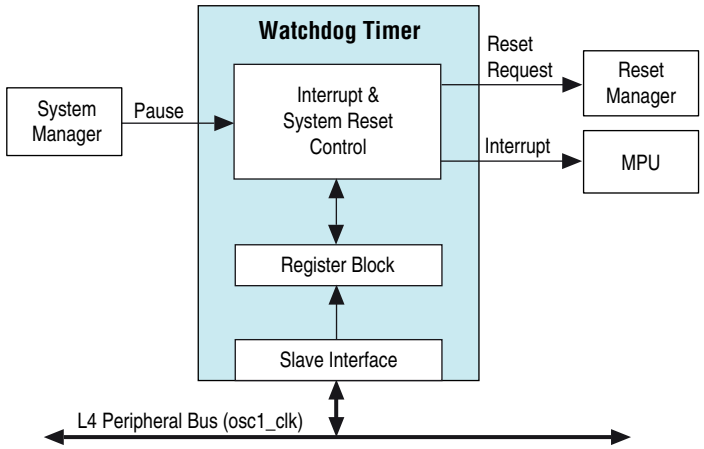
The following list describes the features of the watchdog timer:

- 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

Watchdog Timer Block Diagram and System Integration

Figure 24–1 shows a block diagram of the watchdog timer.

Figure 24–1. Watchdog Timer Block Diagram



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.

For more information, refer to the *Cortex-A9 MPU System* and *Reset Manager* chapters in volume 3 of the *Cyclone V Device Handbook*.

Functional Description of the Watchdog Timer

The watchdog timers are peripherals that can be used to recover from system lockup typically caused by software or system related issues.

Counter

Each watchdog timer is a programmable down counter that decrements by one on each clock cycle. The watchdog timer supports 16 fixed timeout period values and software chooses which timeout periods are desired. A timeout period is 2^n `osc1_clk` clock periods, where n is an integer from 16 to 31 inclusive. For information about programming the timeout period values, refer to "Setting the Timeout Period Values" on page 24–4. For more information about the `osc1_clk` clock, refer to "Clocks" on page 24–3.

If the counter reaches zero, the watchdog timer has timed out, indicating an unrecoverable error has occurred and a system reset is needed. Software must continually 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. For information, refer to "Reloading a Watchdog Counter" on page 24–4.

Software sets the watchdog timer output response mode to either generate a reset request on a timeout, or assert an interrupt request and start counting down a second time. In the former case, the counter wraps and keeps decrementing, even while a reset request is asserted, until the watchdog timer is reset by the reset manager. In the latter case, the generated interrupt is passed to the generic interrupt controller (GIC) in the MPU subsystem. If the interrupt is not serviced by software before a second timeout occurs, the timer generates a reset request. For information about programming the output response mode, refer to "Selecting the Output Response Mode" on page 24–4.

If a restart occurs at the same time the watchdog counter reaches zero, an interrupt is not generated.

Pause Mode

The watchdog timers can be paused during debugging. Pausing the watchdog timers 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 even while debugging.

When the system manager exits reset, the watchdog pausing feature is enabled for both CPUs by default.

For information about programming the pause mode, refer to "Pausing a Watchdog Timer" on page 24–5.

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.

For more information, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Resets

Watchdog timers are reset by a cold or warm reset from the reset manager, and are disabled when exiting reset. †

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Watchdog Timer Programming Model

The watchdog timer is a little-endian module. This section describes the programming options for the timers.

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 steps:

- To set the initial start timeout period, write i to the timeout period for 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.

Set the timeout values before enabling the timer.

Selecting the Output Response Mode

The watchdog timers have two output response modes (as described in "Counter" on page 24–2). To select the desired mode, perform one of the following steps:

- 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` regsiter.

If a restart occurs at the same time the watchdog counter reaches zero, a system reset is not generated. †

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 (`wddbg`) in the system manager.

For more information, refer to register definitions section of the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Disabling and Stopping a Watchdog Timer

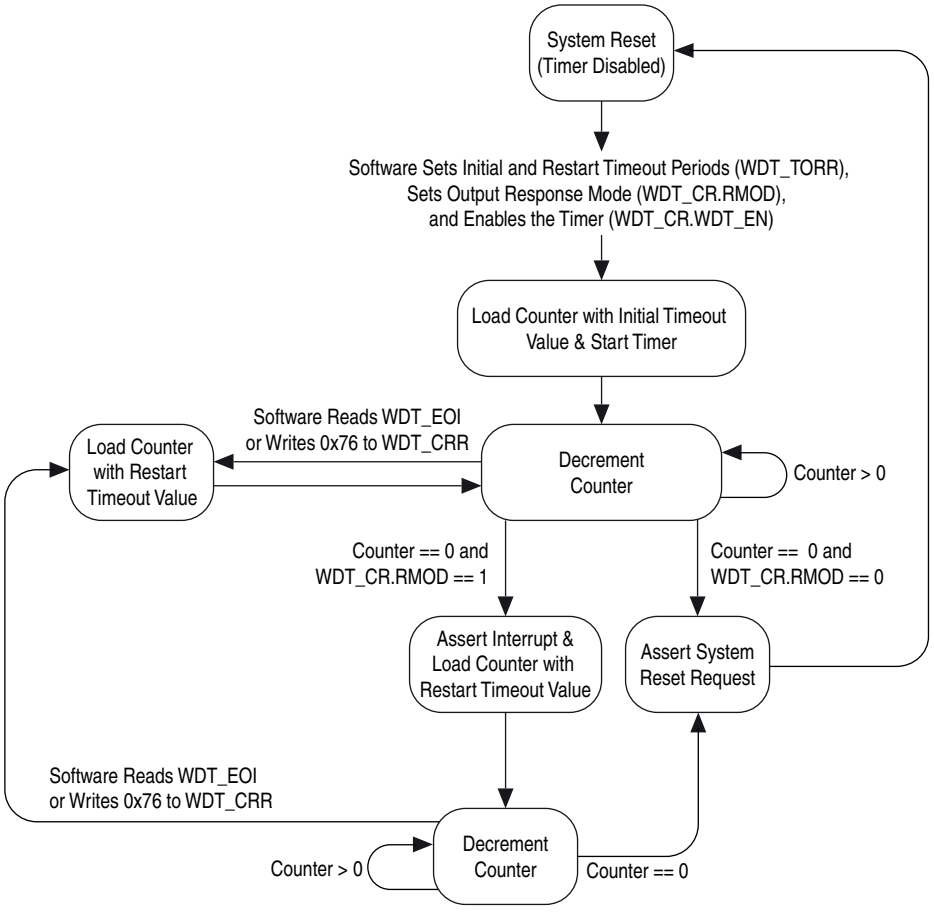
The watchdog timers are disabled and stopped only by resetting them from the reset manager.

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Watchdog Timer State Machine

Figure 24–2 shows a state machine for the watchdog timer functionality.

Figure 24–2. Watchdog Timer State Machine



The state machine 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.

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 reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for either of the following module instances:

- **l4wd0**
- **l4wd1**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 24–1 shows the revision history for this document.

Table 24–1. Document Revision History

Date	Version	Changes
November 2012	1.2	Minor updates.
May 2012	1.1	Added programming model and address map and register definitions sections.
January 2012	1.0	Initial release.

25. CAN Controller

cv_54025-1.2

The hardware processor system (HPS) provides two controller area network (CAN) controllers for serial communication with the Cortex™-A9 microprocessor unit (MPU) subsystem host processor and the direct memory access (DMA) controller using the CAN protocol. The CAN controllers are instances of the Bosch® D_CAN controller and compliant with ISO 11898-1.

Features of the CAN Controller

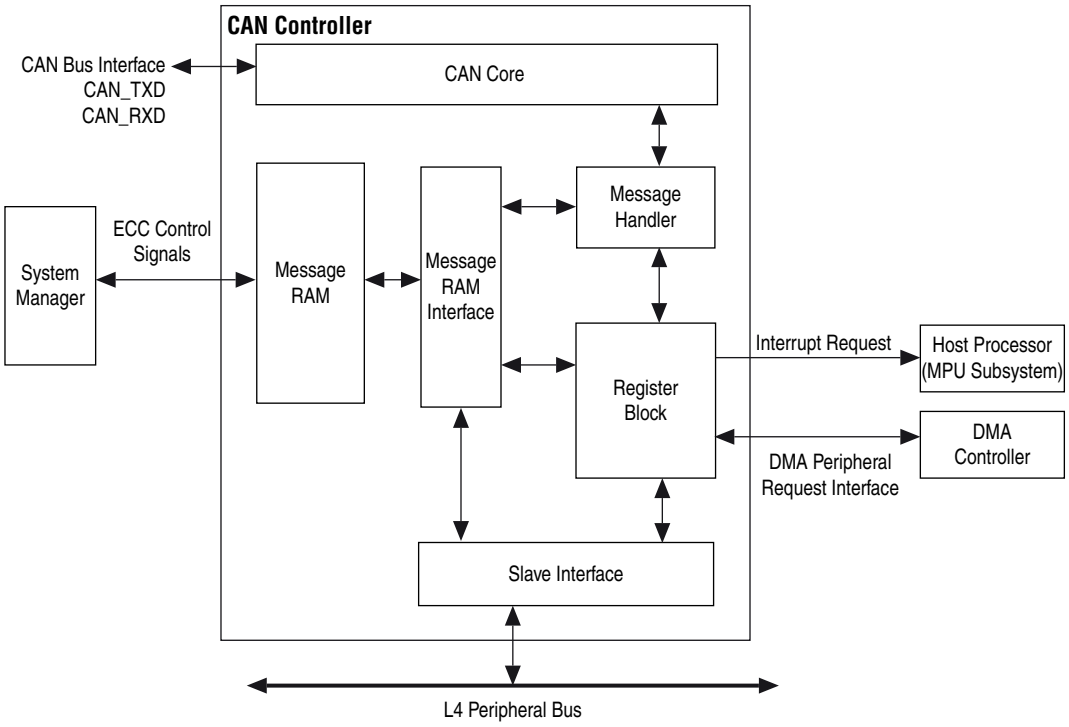
The CAN controllers offer the following features:

- Compliant with *CAN Protocol Specification 2.0* parts A and B, available from the Bosch website (www.semiconductors.bosch.de)
- Programmable communication rate up to 1 Mbps
- Holds up to 128 messages
- Error correction code (ECC)
- 11-bit standard and 29-bit extended identifiers
- Programmable loopback mode
- External direct memory access (DMA) controller for large data transfers
- Automatic retransmission

CAN Controller Block Diagram and System Integration

Figure 25–1 shows the CAN controller block diagram.

Figure 25–1. CAN Controller Block Diagram



The CAN controller consists of the following modules and interfaces:

- CAN core
  - Connects to the CAN bus interface
  - Handles all ISO 11898-1 protocol functions
- Message handler
  - State machine that controls the data transfer between the message RAM and CAN core.
  - Handles acceptance filtering and the interrupt generation
- Message RAM
  - Storage for up to 128 messages objects
  - Single bit error correction and double bit error detection
- Message RAM interface
  - Two separate interfaces, `IF1` and `IF2`

- Register block
  - Control and status registers (CSR) for module setup and indirect message object access.
  - All host processor accesses to the message RAM are relayed through the message RAM interface.
- Level 4 (L4) slave interface for CSR accesses

Functional Description of the CAN Controller

The CAN controllers perform communication according to the CAN protocol version 2.0 parts A and B. All communication on the CAN bus is through message objects. The CAN controller stores message objects in its internal message RAM. The host processor cannot access the message RAM directly, instead the `IF1` and `IF2` message interface register sets provide the host processor access to the messages. Messages are passed between the message RAM and the CAN core by the message handler. The message handler is also responsible for message level responsibilities such as acceptance filtering, interrupt generation, and transmission request generation.

Message Object

The message RAM can store up to 128 message objects. To avoid conflicts between host processor accesses to the message RAM and CAN message reception and transmission, the host processor does not access the message objects directly. Accesses are handled through the `IF1` and `IF2` message interface registers.

Table 25–1 shows the structure of a message object. The first row contains the message object control flags, the second row contains the message object masks, and the third row is the CAN message.

Table 25–1. Message Object Structure

MsgVal				NewDat	MsgLst	IntPnd		TxIE	RxIE	RmtEn	TxRqst	EoB
UMask	Msk [28:0]	MXtd	MDir									
	ID [28:0]	Xtd	Dir	DLC [3:0]	Data 0 [7:0]	Data 1 [7:0]	Data 2 [7:0]	Data 3 [7:0]	Data 4 [7:0]	Data 5 [7:0]	Data 6 [7:0]	Data 7 [7:0]

Message Object Control Flags

This section describes the message object control flags.

Message Valid (`MsgVal`)

- 0= The message object is ignored by the message handler
- 1= The message object is configured and should be considered by the message handler

The host processor must set the `MsgVal` bit of all unused message objects to 0 before it resets the initialization bit (`Init`) of the CAN control register (`CCTRL`) to initialize the CAN controller. `MsgVal` must also be set to 0 when the message object is no longer in use.

The MsgVal field is directly readable from the message valid registers (MOVALA, MOVALB, MOVALC, MOVALD, and MOVALX). However, to write to the MsgVal field for a specific message object, the host processor must write to the message interface registers.

New Data (NewDat)

- 0= No new data has been written into the data portion of this message object by the message handler since last time this flag was cleared by the host processor.
- 1= The message handler or the host processor has written new data to the data portion of this message object.

The NewDat field is directly readable from the new data registers (MONDA, MONDB, MONDC, MONDD, and MONDX). However, to write to the NewDat field for a specific message object, the host processor must write to the message interface registers.

Message Lost (MsgLst)

- 0= No message lost since last time this bit is was last reset by the host processor
- 1= The message handler stored a new message into this object when the NewDat bit was still set, indicating the host processor has lost a message.

MsgLst is valid only in message objects with the message direction bit (Dir) set to receive.

Interrupt Pending (IntPnd)

- 0= This message object is not the source of an interrupt.
- 1= This message object is the source of an interrupt. The interrupt identifier field in the CAN interrupt register (CIR) points to this message object if there is no other interrupt source with higher priority.

The IntPnd field is directly readable from the interrupt pending registers (MOIPA, MOIPB, MOIPC, MOIPD, and MOIPX). However, to write to the IntPnd field for a specific message object, the host processor must write to the message interface registers.

Transmit Interrupt Enable (TxIE)

- 0= TxIE is disabled. IntPnd is left unchanged after the successful transmission of a frame.
- 1= TxIE is enabled. IntPnd is set after a successful transmission of a frame.

Receive Interrupt Enable (RxIE)

- 0= RxIE is disabled. IntPnd is left unchanged after a successful reception of a frame.
- 1= RxIE is enabled. IntPnd is set after a successful reception of a frame.

Remote Enable (RmtEn)

- 0= RmtEn is disabled. At the reception of a remote frame, TxRqst is left unchanged.
- 1= RmtEn is enabled. At the reception of a remote frame, TxRqst is set.

Transmit Request (TxRqst)

- 0= This message object is not waiting for transmission.

- 1= The transmission of this message object is requested and is not complete.

The TxRqst field is directly readable from the transmission request registers (MOTRA, MOTRB, MOTRC, MOTRD, and MOTRX). However, to write to the TxRqst field for a specific message object, the host processor must write to the message interface registers.

End of Block (EoB)

- 0= Message object belongs to a FIFO buffer block and is not the last message object of that FIFO buffer block.
- 1= Single message object or last message object of a FIFO buffer block.

This bit is used to concatenate two or more message objects (up to 128) to build a FIFO buffer. For single message objects (not belonging to a FIFO buffer), this bit must always be set to 1.

Message Object Mask Bits

The message object mask bits, together with the arbitration bits, are used for acceptance filtering of incoming messages.

Use Acceptance Mask (UMask)

- 0= Ignore mask. Msk[28:0], MXtd, and Mdir have no effect on acceptance filtering. For an incoming message to be accepted, all the following conditions must be met:
  - The received message is a data frame with message direction set to 0 (receive) or is a remote frame with message direction set to 1 (transmit).
  - The received message identifier matches the message identifier (ID[28:0]) of the message object.
  - The received identifier extension bit matches the identifier extension bit (Xtd) of the message object.
- 1= Use mask (Msk[28:0], MXtd, and MDir) for acceptance filtering if the respective mask bits are set up for acceptance filtering. For an incoming message to be accepted, all the following conditions must be met:
  - The received message is a data frame with message direction set to 0 (receive) or is a remote frame with message direction set to 1 (transmit) with the MDir mask bit enabled.
  - The received message identifier matches the message identifier (ID[28:0]) of the message object with the Msk[28:0] mask bits enabled
  - The received identifier extension bit matches the identifier extension bit (Xtd) of the message object, with the MXtd mask bit enabled.

If the UMask bit is set to 1, the message object's mask bits have to be programmed during the initialization of the message object before MsgVal is set to 1.

Identifier Mask (Msk[28:0])

The identifier mask filters the corresponding bits in ID[28:0].

- 0= The corresponding identifier bit has no effect on the acceptance filter.
- 1= The corresponding identifier bit is used for acceptance filtering.

Extended Identifier Mask (MXtd)

- 0= The extended frame identifier bit (Xtd) has no effect on the acceptance filtering
- 1= The extended frame identifier bit (Xtd) is used for acceptance filtering

When 11-bit (standard) identifiers are used for a message object, the identifiers of received data frames are written to bits ID28 to ID18. For acceptance filtering, only these bits, together with mask bits Msk28 to Msk18, are used.

Mask Message Direction (MDir)

- 0= The message direction bit (Dir) has no effect on the acceptance filtering.
- 1= The message direction bit (Dir) is used for acceptance filtering.

(1) Altera recommends always setting MDir to 1. Ignoring the message direction bit is an advanced technique that must be handled with great care.

CAN Message Bits

The arbitration fields ID28-0, Xtd, and Dir are used to define the identifier and type of outgoing messages and are used (together with the mask fields Msk28-0, MXtd, and MDir) for acceptance filtering of incoming messages. A received message is stored to the valid message object with matching identifier when the direction is set to receive a data frame or transmit a remote frame. Extended frames can be stored only in message objects with Xtd is set to 1, standard frames in message objects with Xtd is set to 0. If a received message (data frame or remote frame) matches more than one valid message object, it is stored to the object with the lowest message number.

Message Identifier (ID[28:0])

- ID28-ID0: 29-bit identifier (extended frame)
- ID28-ID18: 11-bit identifier (standard frame)

Extended Frame Identifier (Xtd)

- 0= The 11-bit (standard) identifier is used for this message object.
- 1= The 29-bit (extended) identifier is used for this message object.

Message Direction (Dir)

- 0= receive direction. When TxRqst is set to 1, a remote frame with the identifier of this message object is transmitted. On reception of a data frame with matching identifier, that message is stored in this message object.
- 1= transmit direction. When TxRqst is set to 1, the respective message object is transmitted as a data frame. On reception of a remote frame with matching identifier, the TxRqst bit of this message object is set to 1 (if RmtEn = 1).

Data Length Code (DLC[3:0])

DLC specifies the number of data bytes in the data frame. The maximum number is eight.

The data length code (DLC) of a message object must be defined the same as in all the corresponding objects with the same identifier in all CAN devices. When the message handler stores a data frame, it sets the DLC field to the value provided in the received message.

Data Bytes 0-7 (`Data 0[7:0]` - `Data 7[7:0]`)

The data bytes in a CAN data frame.

Message Interface Registers

There are two sets of message interface registers, `IF1` and `IF2`, that provide a means for a host processor or DMA controller to access any message object indirectly. Message objects are transferred between the message RAM and the message buffer registers as a single, atomic operation maintaining data consistency across the message.

Table 25–2 lists the structure of each message interface register set, where *x* represents either set 1 or set 2.

Table 25–2. Message Interface Register Set

Register Type	Register	Name	Description
Command	`IFxCMR`	IFx command register	Specifies the transfer direction and selects the portions of the message object to transfer
Message buffer	`IFxMSK`	IFx mask register	Provides access to the `Msk`, `MDir`, and `MXtd` mask fields of the message object
	`IFxARB`	IFx arbitration register	Provides access to the `ID`, `Dir`, `Xtd`, and `MsgVal` arbitration fields of the message object
	`IFxMCTR`	IFx message control register	Provides access to the `DLC`, `EoB`, `TxRqst`, `RmtEn`, `RxIE`, `TxIE`, `UMask`, `IntPnd`, `MsgLst`, and `NewDat` fields of the message object
	`IFxDA`	IFx data A register	Provides access to data bytes 0-3 of the message object
	`IFxDB`	IFx data B register	Provides access to data bytes 4-7 of the message object

DMA Mode

The CAN controller, shown in Figure 25–1, can issue DMA controller requests to transfer data between one or both of the message interface registers and system memory. The CAN controller has two DMA request interfaces, called `can_if1dma` and `can_if2dma`. The CAN peripheral request interfaces are shared with the FPGA DMA peripheral request interfaces. To use the DMA peripheral request interface, the host processor must access the CAN control register (`CCTRL`) in the protocol group (`protogrp`). The peripheral request interface is selected through the system manager.

For more information about the selecting the CAN DMA peripheral request interface, refer to the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

To activate the DMA support feature and initiate a transfer, write a 1 to the `DMAactive` bit in the appropriate IF command register (`IFxCMR`) in the message interface group (`msgifgrp`). After the message object transfer is completed, the CAN controller issues a DMA peripheral request to perform the next message object transfer. The request remains active until the first read or write to the message interface register.

For more information, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

Automatic Retransmission

The CAN controller provides a means for automatic retransmission of frames that have lost arbitration or have errors during transmission. Retransmission happens automatically without user intervention or notification. Normal confirmation is given when the transmission is successfully complete.

Test Mode

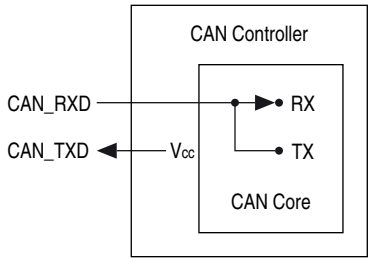
To enable test mode, set the test mode enable bit (`Test`) in the `CCTRL` register to 1. This action activates write access to the CAN test register (`CTR`). The following sections describe the available test modes.

Silent Mode

The CAN controller is set in the silent mode by programming the silent mode (`Silent`) bit in the test register (`CTR`) to 1. In silent mode, the CAN controller is able to receive valid data frames and valid remote frames, but it holds the `CAN_TXD` pin high, sending no data to the CAN bus. The silent mode can be used to analyze the traffic on a CAN bus without affecting it by the transmission of dominant bits (acknowledge bits, error frames). In ISO 11898-1, the silent mode is called the *bus monitoring mode*.

Figure 25–2 shows the CAN core in silent mode.

Figure 25–2. CAN Core in Silent Mode



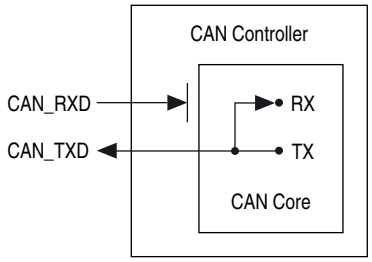
Loopback Mode

The CAN controller is set in loopback mode by programming the loopback mode (`LBack`) bit in the test register (`CTR`) to 1. In loopback mode, the CAN controller treats its own transmitted messages as received messages and stores them (if they pass acceptance filtering) to a receive buffer.

To be independent from external stimulation, the CAN controller ignores acknowledge errors in loopback mode. In this mode, the CAN controller provides internal feedback from its transmit (TX) output to its receive (RX) input. The actual value of the input pin is disregarded by the CAN controller.

Figure 25–3 shows the CAN core in loopback mode.

Figure 25–3. CAN Core in Loopback Mode

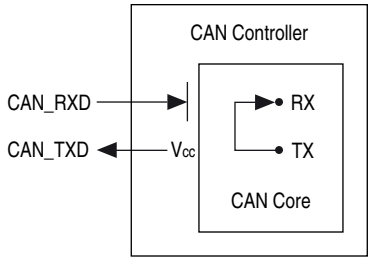


Combined Mode

The CAN controller is set in combined loopback and silent mode by programming the silent mode (`Silent`) and loopback mode (`LBack`) bits in the test register (`CTR`) to 1. Combined mode can be used for testing the CAN hardware without affecting other devices connected to the CAN bus. In this mode, the `CAN_RXD` pin is disconnected from the CAN core and the `CAN_TXD` pin is held high.

Figure 25–4 shows the CAN core in combined mode.

Figure 25–4. CAN Core in Combined Mode



L4 Slave Interface

The host processor accesses data, control, and status information of the CAN controller through the L4 slave interface. The slave interface supports 32-bit accesses only.

This interface does not support error responses.

Clocks

The CAN controller operates on the `l4_sp_clk` and `can_clk` clock inputs. The `l4_sp_clk` clock is used by the L4 slave interface and the `can_clk` is used to operate the CAN core.

The `can_clk` clock must be programmed to be at least eight times the CAN bus interface speed. For example, for a CAN bus interface operating at a 1 Mbps baud rate, the `can_clk` clock must be set to at least 8 MHz. The `l4_sp_clk` clock can operate at a clock frequency that is equal to or greater than the `can_clk` frequency.

For more information about the l4_sp_clk and can_clk clocks, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Resets

Each CAN controller can be reset by software or by hardware.

Software Reset

Software initialization is done by setting the Init bit in the CAN control register (CCTRL) in the protocol group (protogrp) in the CAN controller register map. This bit is set through the CAN protocol when a bus off condition occurs on the CAN link. The bit is also set through the hardware reset input described in "Hardware Reset".

Due to the synchronization mechanism between the two clock domains, there might be a delay until the value written to the Init bit can be read back. To assure that the previous value written has been accepted, read the Init bit before setting it to a new value.

The bus off recovery sequence cannot be shortened by setting or resetting the Init bit. For more information about bus off, refer to the *CAN Protocol Specification 2.0* parts A and B, available from the Bosch website (www.semiconductors.bosch.de).

Hardware Reset

Each CAN controller has a separate reset signal. The reset manager drives the signals on a cold or warm reset. The reset signal is synchronized to both clock domains and applied to the appropriate logic within the CAN controllers.

For more information, refer to the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Interrupts

Each CAN controller generates two interrupt signals. One signal indicates error and status interrupts and the other signal indicates message object interrupts. Both interrupt signals connect to the global interrupt controller (GIC). Interrupts are enabled in the CAN control register (CCTRL) in the protocol group (protogrp). The CAN interrupt register (CIR) in the protocol group (protogrp) indicates the highest priority interrupt that is pending.

Error Interrupts

The following error conditions generate interrupts:

- Bus off—when the transmit error count is equal to or greater than 256, the bus off (BOff) bit in the CAN status register (CSTS) in the protocol group (protogrp) is set to 1.
- Error warning—when either the transmit error counter or the receive error counters reaches 96, the error warning status (EWarn) bit in the CAN status register (CSTS) in the protocol group (protogrp) is set to 1.

Status Interrupts

The following status conditions generate interrupts:

- Receive OK—when the CAN controller receives a message successfully, the RxOK bit in the CAN status register (CSTS) in the protocol group (protogrp) is set to 1.
- Transmit OK—when the CAN controller transmits a message successfully, the TxOK bit in the CAN status register (CSTS) in the protocol group (protogrp) is set to 1.
- Last error code—when a message is received or transmitted with an error, the LEC bits in the CAN status register (CSTS) in the protocol group (protogrp) are set according to the error type.

Message Object Interrupts

The IntPnd bits from the message objects can generate interrupts when the corresponding message object TxIE bit or RxIE bit is set to 1. Table 25–2 lists the location of message object interrupt information in the interrupt pending registers. The interrupt pending registers are located in the message handler group (msghandgrp).

Table 25–3. Message Object Interrupt Registers

Register	Title	Message Objects
MOIPA	Interrupt pending A register	1 to 32
MOIPB	Interrupt pending B register	33 to 64
MOIPC	Interrupt pending C register	65 to 96
MOIPD	Interrupt pending D register	97 to 128

The MOIPX register allows software to quickly detect which message object group has a pending interrupt.

CAN Controller Programming Model

This section describes how to operate the CAN controllers.

Software Initialization

The software initialization is started by setting the Init bit in the CAN control register (CCTRL) to 1. While the Init bit is 1, messages are not transferred to or from the CAN bus, and the CAN_TXD CAN bus output is held in the high state. Setting the Init bit does not change any configuration registers.

To initialize the CAN controller, the host processor must program the CAN bit timing (CBT) register and message objects that will be used for CAN communication. If a message object is not needed, it is sufficient to set the MsgVal bit of the message object to not valid (0), which is the default after RAM initialization. You must set up the entire message object before setting MsgVal bit to valid (1). The message objects are set up through either message interface register set.

Access to the CAN bit timing (CBT) register is only enabled when the configuration change enable (CCE) and Init bits in the CAN control register (CCTRL) are both set to 1.

Setting the Init bit to 0 finishes the software initialization. The CAN core synchronizes itself to the data transfer on the CAN bus by waiting for the bus to reach an idle state before it can take part in bus activities and message transfers.

The initialization of the message objects is independent of the CAN controller initialization and can be done anytime, but the message objects should all be configured to particular identifiers or set to not valid before message transferring begins.

On power up, the message RAM has to be initialized. To initialize RAM, set the Init bit to 1, then set the RAMInit bit in the CAN function register (CFR) in the protocol group (protogrp) to 1. The RAMInit bit returns to 0 when RAM initialization completes. During RAM initialization, all message objects are cleared to zero and the RAM ECC bits are initialized. Access to RAM is not allowed prior to or during RAM initialization.

CAN Message Transfer

Once the CAN controller is initialized, the CAN controller synchronizes itself to the CAN bus and starts transferring messages.

Received messages are stored to their appropriate message objects, if they pass the message handler's acceptance filtering. The entire message, including all arbitration bits, Xtd, Dir, DLC, eight data bytes, the mask and control bits UMask, MXtd, MDir, EoB, MsgLst, RxIE, TxIE, and RmtEn, is stored in the message object. Masked arbitration bits might change in the message object when a received message is stored.

The host processor may read or update each message at any time using the message interface registers. The message handler guarantees data consistency when the host processor accesses the message object at the same time the message is being transferred to or from RAM.

Messages to be transmitted are updated by the host processor. If a permanent message object (arbitration and control bits set up during configuration are unchanged for multiple CAN transfers) exists for the message, only the data bytes need to be updated. If several transmit messages are assigned to the same message object (when the number of message objects is not sufficient), the whole message object has to be configured before the transmission of this message is requested.

The transmission of any number of message objects may be requested at the same time they are transmitted, according to their internal priority. The message object numbers are 1 to 128, with 1 being the lowest internal priority and 128 the highest priority. Messages may be updated or set to invalid (MsgVal=0) at anytime, even when their requested transmission is still pending. The old data is discarded when a message is updated before its pending transmission has started.

Depending on the configuration on the message object, the transmission of a message may be requested automatically by the reception of a remote frame with a matching identifier. Remote frames are frames used to request a particular message on the CAN network.

For ease in programming, Altera recommends using one IF message interface for all receive direction activity and the other IF message interface for all transmit direction activity.

Message Object Reconfiguration for Frame Reception

To configure a message object to receive data frames, set the Dir field to 0.

To configure a message object to receive remote frames, set the Dir field to 1, set UMask to 1, and set RmtEn to 0.

To avoid modifying an object while it is being transmitted, you must set MsgVal to 0 before changing any of the following configuration and control bits:

- ID[28:0]
- Xtd
- DLC[3:0]
- RxIE
- TxIE
- RmtEn
- EoB
- UMask
- Msk[28:0]
- MXtd
- MDir

The following fields of a message object can be changed without clearing MsgVal:

- Data0[7:0] to Data7[7:0]
- TxRqst
- NewDat
- MsgLst
- IntPnd

Message Object Reconfiguration for Frame Transmission

To configure a message object to transmit data frames, set the Dir field to 1, and either set UMask to 0 or set RmtEn to 1.

Before changing any of the following configuration and control bits, you must set MsgVal to 0:

- Dir
- RxIE
- TxIE
- RmtEn
- EoB
- UMask
- Msk[28:0]

- MXtd
- MDir

The following fields of a message object can be changed without clearing MsgVal:

- ID[28:0]
- Xtd
- DLC[3:0]
- Data0[7:0] to Data7[7:0]
- TxRqst
- NewDat
- MsgLst
- IntPnd

CAN Controller Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the module description and base address, scroll to and click the link for either of the following module instances:

- **can0**
- **can1**

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 25–4 shows the revision history for this document.

Table 25–4. Document Revision History

Date	Version	Changes
November 2012	1.2	■ Minor updates. ■ Expanded reset section. ■ Expanded interrupts seciton.
May 2012	1.1	Added block diagram and system integration, functional description, programming model, and address map and register definitions sections.
January 2012	1.0	Initial release.

Section VII. Hard Processor System User Guide

This section includes the following chapters:

- Chapter 26, Introduction to the HPS Component
- Chapter 27, Instantiating the HPS Component
- Chapter 28, HPS Component Interfaces
- Chapter 29, Simulating the HPS Component

For information about the revision history for chapters in this section, refer to "Document Revision History" in each individual chapter.

26. Introduction to the HPS Component

cv_54026-1.0

The hard processor system (HPS) component is a soft component that you can instantiate in the FPGA fabric of the Cyclone® V SoC FPGA. It 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.

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 HPS system architecture and features, refer to the *Introduction to the Hard Processor* chapter in volume 3 of the *Cyclone V Device Handbook* and the *CoreSight Debug and Trace* chapter in volume 3 of the *Cyclone V Device Handbook*.

For descriptions of individual peripheral architectures and features, refer to the following chapters and sections:

- "HPS Block Diagram and System Integration" in the *Introduction to the Hard Processor* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "Clock Manager Block Diagram and System Integration" in the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "Reset Manager Block Diagram and System Integration" in the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "Interconnect Block Diagram and System Integration" in the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "AXI Bridges Block Diagrams and System Integration" in the *HPS-FPGA AXI Bridges* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "Cortex-A9 MPU Subsystem Block Diagram and System Integration" in the *Cortex-A9 Microprocessor Unit Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "CoreSight Debug and Trace Block Diagram and System Integration" in the *CoreSight Debug and Trace* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "SDRAM Controller Subsystem Block Diagram and System Integration" in the *SDRAM Controller Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "On-Chip RAM Block Diagram and System Integration" in the *On-Chip Memory* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "Boot ROM Block Diagram and System Integration" in the *On-Chip Memory* chapter in volume 3 of the *Cyclone V Device Handbook*.

- "NAND Flash Controller Block Diagram and System Integration" in the *NAND Flash Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "SD/MMC Controller Block Diagram and System Integration" in the *SD/MMC Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "Quad SPI Flash Controller Block Diagram and System Integration" in the *Quad SPI Flash Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "FPGA Manager Block Diagram and System Integration" in the *FPGA Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "System Manager Block Diagram and System Integration" in the *System Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "Scan Manager Block Diagram and System Integration" in the *Scan Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "DMA Controller Block Diagram and System Integration" in the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "EMAC Block Diagram and System Integration" in the *Ethernet Media Access Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "USB OTG Controller Block Diagram and System Integration" in the *USB 2.0 OTG Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "SPI Block Diagram and System Integration" in the *SPI Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "I²C Controller Block Diagram and System Integration" in the *I²C Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "UART Controller Block Diagram and System Integration" in the *UART Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "General-Purpose I/O Interface Block Diagram and System Integration" in the *General-Purpose I/O Interface* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "Timer Block Diagram and System Integration" in the *Timer* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "Watchdog Timer Block Diagram and System Integration" in the *Watchdog Timer* chapter in volume 3 of the *Cyclone V Device Handbook*.
- "CAN Controller Block Diagram and System Integration" in the *CAN Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link to open the file.

To view the description and base address for a specific peripheral, scroll to and click the link for the peripheral's module name.

To then view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

Document Revision History

Table 26–1 shows the revision history for this document.

Table 26–1. Document Revision History

Date	Version	Changes
June 2012	1.0	Initial release.
May 2012	0.1	Preliminary draft.

27. Instantiating the HPS Component

cv_54027-1.1

You instantiate the hard processor system (HPS) component in Qsys. The HPS is available in the component library under **Embedded Processors**. This chapter describes the parameters available in the HPS component parameter editor, which opens when you add or edit an HPS component.

The HPS requires specific device targets. For a detailed list of supported devices, refer to the *Cyclone® V Device Datasheet*.

For general information about using Qsys, refer to the *Creating a System with Qsys* chapter in volume 1 of the *Quartus® II Handbook*.

Configuring FPGA Interfaces

This section describes parameters on the **FPGA Interfaces** tab.

For general information about interfaces, refer to the *HPS Component Interfaces* chapter in volume 3 of the *Cyclone V Device Handbook*.

General Interfaces

This section describes parameters in the **General** group on the **FPGA Interfaces** tab. When enabled, the interfaces described in Table 27–1 become visible in the HPS component.

Table 27–1. 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. ■ Wake up an MPCore processor from a wait for event (WFE) state.	h2f_mpu_events
Enable MPU 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_mpu_gp
Enable FPGA-to-HPS Interrupts	Enables interface for FPGA interrupt signals to the MPU (in the HPS).	f2h_irq0 f2h_irq1
Enable Debug APB interface	Enables debug interface to the FPGA, allowing access to debug components in the HPS. (1)	h2f_debug_apb h2f_debug_apb_sideband h2f_debug_apb_clock
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. (1)	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). (1)	h2f_cti h2f_cti_clock
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. (1)	h2f_tpiu h2f_tpiu_clock_in

Note to Table 27–1:

(1) For information about this functionality, refer to the *CoreSight Debug and Trace* chapter in volume 3 of the Cyclone V Device Handbook.

Boot and Clock Selection Interfaces

This section describes parameters in the **Boot and Clock Selection** group in the **FPGA Interfaces** tab.

Table 27–2 lists the available parameters.

Table 27–2. Boot and Clock Selection Parameters

Parameter Name	Parameter Description
Enable boot from FPGA ready	Enables an input to the HPS indicating whether a preloader is available in on-chip RAM. If the input is asserted, a preloader image is ready at memory location 0.
Enable boot from FPGA on failure	Enables an input to the HPS indicating whether a fallback preloader is available in on-chip RAM. If the input is asserted, a fallback preloader image is ready at memory location 0. The fallback preloader is to be used only if the HPS boot ROM does not find a valid preloader image in the selected flash memory device.

For detailed information about the HPS boot sequence, refer to the *Booting and Configuration* appendix in volume 3 of the *Cyclone V Device Handbook*.

AXI Bridges

This section describes parameters in the **AXI Bridges** group on the **FPGA Interfaces** tab.

Table 27–3. 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`
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`
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`

To facilitate accessing these slaves from a memory-mapped master with a smaller address width, you can use the Altera® Address Span Extender. The address span extender is discussed in "Using the Address Span Extender Component" on page 27–9.

For more information, refer to the *Interconnect* chapter in volume 3 of the *Cyclone V Device Handbook*.

FPGA-to-HPS SDRAM Interface

This section describes parameters in the **FPGA-to-HPS SDRAM Interface** group on the **FPGA Interfaces** tab.

You can add one or more SDRAM ports that make the HPS SDRAM subsystem accessible to the FPGA fabric.

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 Altera Address Span Extender. The address span extender is discussed in "Using the Address Span Extender Component" on page 27–9.

On the **FPGA Interfaces** tab, in the **FPGA to HPS SDRAM Interface** table, use **+** or **–** to add or remove FPGA-to-HPS SDRAM interfaces. The **Name** column denotes the interface name. Table 27–4 shows the parameters available for each SDRAM interface.

Table 27–4. FPGA-to-HPS SDRAM Interface Parameters

Parameter Name	Parameter Description
Name	Port name (auto assigned as shown in Table 27–5)
Type	Interface type: ■ AXI-3 ■ Avalon-MM Bidirectional ■ Avalon-MM Write-only ■ Avalon-MM Read-only
Width	32, 64, 128, or 256

Table 27–5. FPGA-to-HPS SDRAM Port and Interface Names

Port Name	Interface Name
f2h_sdram0	`f2h_sdram0_data`
f2h_sdram1	`f2h_sdram1_data`
f2h_sdram2	`f2h_sdram2_data`
f2h_sdram3	`f2h_sdram3_data`
f2h_sdram4	`f2h_sdram4_data`
f2h_sdram5	`f2h_sdram5_data`

For more information, refer to the *SDRAM Controller Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*.

Reset Interfaces

This section describes parameters in the **Resets** group on the **FPGA Interfaces** tab. You can enable most resets on an individual basis. Table 27–6 lists the available reset parameters.

Table 27–6. 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`
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`

For more information about the reset interfaces, refer to "Functional Description of the Reset Manager" in the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

DMA Peripheral Request

This section describes parameters in the **DMA Peripheral Request** group on the **FPGA Interfaces** tab.

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.

Peripheral request IDs 4–7 are shared with the controller area network (CAN) controllers.

For more information, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

Configuring Peripheral Pin Multiplexing

This section describes parameters on the **Peripheral Pin Multiplexing** tab.

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 a mode settings specific to their functions.

Each list in the **Peripheral Pin Multiplexing** tab has a hint describing in detail the options available in the list. The hint for each pin multiplexing list shows the I/O pins used by each available pin set. The hint for each mode list shows the signals used by each available mode.

View each hint by hovering over the corresponding list.

The tooltips for the combo boxes show useful information. The pin multiplexing parameters have tables showing the pin destinations and the mode parameters show which signals are used in which mode.

You can enable the following types of peripherals. For details of peripheral-specific settings, refer to the chapter for each peripheral:

- Ethernet Media Access Controller—*Ethernet Media Access Controller* chapter in volume 3 of the *Cyclone V Device Handbook*
- NAND Flash Controller—*NAND Flash Controller* chapter in volume 3 of the *Cyclone V Device Handbook*
- Quad serial peripheral interface (SPI) Flash Controller—*Quad SPI Flash Controller* chapter in volume 3 of the *Cyclone V Device Handbook*
- Secure Digital / MultiMediaCard (SD/MMC) Controller—*SD/MMC Controller* chapter in volume 3 of the *Cyclone V Device Handbook*
- USB 2.0 On-The-Go (OTG) Controllers—*USB 2.0 OTG Controller* chapter in volume 3 of the *Cyclone V Device Handbook*
- SPI Controllers—*SPI Controller* chapter in volume 3 of the *Cyclone V Device Handbook*
- UART Controllers— *UART Controller* chapter in volume 3 of the *Cyclone V Device Handbook*
- Inter-integrated circuit (I^2C) Controllers— *I2C Controller* chapter in volume 3 of the *Cyclone V Device Handbook*
- CAN Controllers—*CAN Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.
- Trace port interface unit (TPIU)—*CoreSight Debug and Trace* chapter in volume 3 of the *Cyclone V Device Handbook.* Enabling the TPIU exposes trace signals to the device pins.

Connecting Unassigned Pins to GPIO

On the **Peripheral Pin Multiplexing** tab, the **Conflicts** table shows pins that are not assigned to any peripheral. By default, general-purpose I/O (GPIO) is disabled on these pins. You can enable a pin as GPIO by changing the **GPIO Enabled** field in the table. The table also shows the GPIO pin number assigned to that pin.

Resolving Pin Multiplexing Conflicts

Use the **Conflicts** table to view pins with invalid multiple assignments. The table shows one of two or more peripheral interfaces assigned to the same pin(s). You can use the peripherals' pin configuration to determine which other peripheral(s) are in conflict, and to resolve the conflict.

For detailed information about available HPS pin configurations, refer to the *Cyclone V Device Family Pin Connection Guidelines*.

Configuring HPS Clocks

This section describes parameters on the **HPS Clocks** tab.

For general information about clock signals, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

User Clocks

This section describes parameters in the **User Clocks** group on the **HPS Clocks** tab.

When you enable a user clock, you must manually enter its maximum frequency for timing analysis. The TimeQuest 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.

Table 27–7 lists the 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" on page 27–9.

Table 27–7. 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	

The clock frequencies you provide are reported in a Synopsys Design Constraints File (**.sdc**) generated by Qsys.

For details about driving these clocks, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

PLL Reference Clocks

This section describes parameters in the **PLL Reference Clocks** group on the **HPS Clocks** tab.

Table 27–8. 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`

For more information, refer to the *Clock Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Configuring the External Memory Interface

This section describes parameters on the **SDRAM** tab.

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 Altera SDRAM controller. 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 **PHY Settings** tab 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 Altera's 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 II 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".

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:

- The *Implementing and Parameterizing Memory IP* chapter in volume 2 of the *External Memory Interface Handbook*.
- The *Functional Description—Hard Memory Interface* chapter in volume 3 of the *External Memory Interface Handbook*. "EMI-Related HPS Features in SoC Devices" describes features specific to the HPS SDRAM controller.

Selecting PLL Output Frequency and Phase

You select PLL output frequency and phase with controls in the **PHY Settings** tab in the **SDRAM** 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 II 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 TimeQuest 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**.

If you turn on **Use specified frequency instead of calculated frequency**, the Quartus II software assumes that the value in the **Achieved memory clock frequency** box is correct. If it is not, timing analysis results are incorrect.

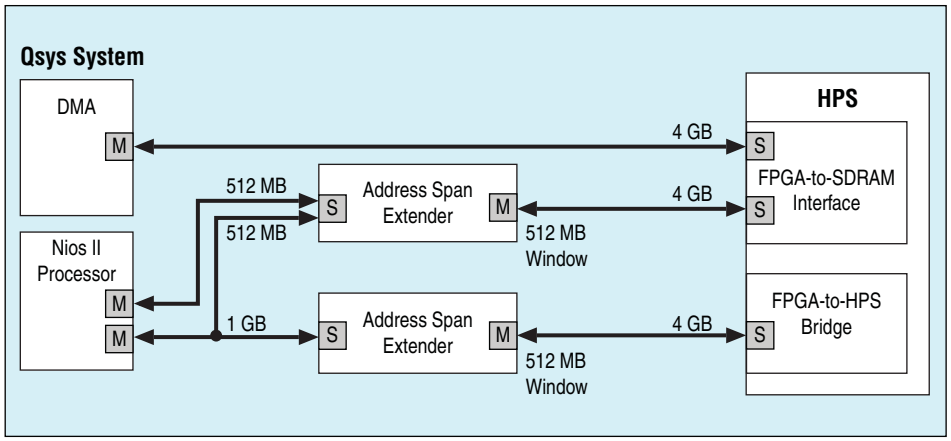
Using the Address Span Extender Component

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 27–1 shows how two address span extender components might be used in a system with the HPS.

Figure 27–1. Address Span Extender



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.

For more information about the address span extender, refer to "Bridges" in the *Qsys Interconnect and System Design Components* chapter in volume 1 of the *Quartus II 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 Qsys project. Perform the following steps:

1. Generate the design with Qsys. The generated files include an **.sdc** file containing clock timing constraints. If simulation is enabled, simulation files are also generated.

   For information about generating a Qsys project, refer to the *Creating a System with Qsys* chapter in volume 1 of the *Quartus II Handbook*. For a description of the simulation files generated, refer to "Simulation Flow" in the *Simulating the HPS Component* chapter in volume 3 of the *Cyclone V Device Handbook*.

2. Add **system.qip** to the Quartus II project. **system.qip** is the Quartus II IP File for the HPS component, generated by Qsys.

3. Perform Analysis and Elaboration with the Quartus II software.
4. Assign constraints to the SDRAM component. When Qsys generates the HPS component (step 1), it generates the pin assignment Tcl Script File (**.tcl**) to perform memory assignments. The script file name is *<qsys_system_name>*_**pin_assignments.tcl**, where *<qsys_system_name>* is the name of your Qsys system. Run this script to assign constraints to the SDRAM component.

   For information about running the pin assignment script, refer to "MegaWizard Plug-In Manager Flow" in the *Implementing and Parameterizing Memory IP* chapter in volume 2 of 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" on page 27–5, 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 Qsys design, with information enabling the Quartus II 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 II software.
6. Optionally back-annotate the SDRAM pin assignments, to eliminate pin assignment warnings the next time you compile the design.

   For information about back-annotating pin assignments, refer to *About Back-Annotating Assignments* in Quartus II Help.

Document Revision History

Table 27–9 shows the revision history for this document.

Table 27–9. Document Revision History

Date	Version	Changes
November 2012	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
June 2012	1.0	Initial release.
May 2012	0.1	Preliminary draft.

28. HPS Component Interfaces

cv_54028-1.1

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 TimeQuest to accurately estimate system timing. It has no effect on PLL settings.

For information about instantiating the HPS component, refer to the *Instantiating the HPS Component* chapter in volume 3 of the *Cyclone® V Device Handbook*. For Avalon™ protocol timing, refer to *Avalon Interface Specifications*. For Advanced Microcontroller Bus Architecture (AMBA®) Advanced eXtensible Interface (AXI™) protocol timing, refer to the *AMBA AXI Protocol Specification v1.0*, which you can download from the ARM website (infocenter.arm.com).

Memory-Mapped Interfaces

FPGA-to-HPS Bridge

Table 28–1. FPGA-to-HPS Bridges and Clocks

Interface Name	Description	Associated Clock Interface (1)
`f2h_axi_slave`	FPGA-to-HPS AXI slave interface	`f2h_axi_clock`

Note to Table 28–1:

(1) Refer to "Clocks" on page 28–4 for information about clock interfaces.

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.

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 adaptors. The Qsys system integration tool automatically generates adaptor 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 Altera® Address Span Extender.

For more information about the FPGA-to-HPS bridge, refer to the *HPS-FPGA AXI Bridges* chapter in volume 3 of the *Cyclone V Device Handbook*. For information about the address span extender, refer to "Using the Address Span Extender Component" in the *Instantiating the HPS Component* chapter in volume 3 of the *Cyclone V Device Handbook*.

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.

For more information about the ACP sideband signals, refer to the *Cortex-A9 Microprocessor Unit Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*.

HPS-to-FPGA and Lightweight HPS-to-FPGA Bridges

Table 28–2. HPS-to-FPGA and Lightweight HPS-to-FPGA Bridges and Clocks

Interface Name	Description	Associated Clock Interface (1)
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

Note to Table 28–2:

(1) Refer to "Clocks" on page 28–4 for information about clock interfaces.

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 adaptors. The Qsys 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.

For more information, refer to the *HPS-FPGA AXI Bridges* chapter in volume 3 of the *Cyclone V Device Handbook*.

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 28–3. HPGA-to-HPS SDRAM Interfaces and Clocks

Interface Name	Description	Associated Clock Interface (1)
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

Note to Table 28–3:

(1) Refer to "Clocks" on page 28–4 for information about clock interfaces.

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 Altera Address Span Extender.

For more information about available combinations of interfaces and ports, refer to the *SDRAM Controller Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*. For information about the address span extender, refer to "Using the Address Span Extender Component" in the *Instantiating the HPS Component* chapter in volume 3 of the *Cyclone V Device Handbook*.

Clocks

The HPS-to-FPGA clock interface supplies physical clocks and resets 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

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

Resets

This section describes the reset interfaces to the HPS component.

For details about the HPS reset sequences, refer to "Functional Description of the Reset Manager" in the *Reset Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

HPS-to-FPGA Reset Interfaces

The following interfaces allow the HPS to reset soft logic in the FPGA fabric:

- `h2f_reset`—HPS-to-FPGA cold and 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 Sources

The following interfaces allow soft logic in the FPGA fabric to reset the HPS:

- `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

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`

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 Peripheral Request Interfaces

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

For more information, refer to the *DMA Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

Other Interfaces

MPU Standby and Event Interfaces

MPU standby and event 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 standby and event signals are included in the following interfaces:

- `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-A9 processors.
  - `h2f_mpu_standbywfe[1:0]`—Indicates whether each Cortex-A9 processor is in the WFE state
  - `h2f_mpu_standbywfi[1:0]`—Indicates whether each Cortex-A9 processor is in the wait for interrupt (WFI) state
- `h2f_mpu_gp`—General purpose interface

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.

For more information, refer to the *Cortex-A9 Microprocessor Unit Subsystem* chapter in volume 3 of the *Cyclone V Device Handbook*.

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`).

General-Purpose Interfaces

You can use the FPGA manager to supply the h2f_mpu_gp interface, which includes the following general-purpose signals:

- 32 FPGA-to-HPS signals
- 32 HPS-to-FPGA signals

For more information, refer to the *FPGA Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table 28–4 shows the revision history for this document.

Table 28–4. Document Revision History

Date	Version	Changes
November 2012	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 Altera Address Span Extender.
June 2012	1.0	Initial release.
May 2012	0.1	Preliminary draft.

29. Simulating the HPS Component

cv_54030-1.1

HPS Simulation Support

This section describes the simulation support for the hard processor system (HPS) component. The HPS simulation models support interfaces between the HPS and FPGA fabric, including:

- Bus functional models (BFMs) for most interfaces between HPS and FPGA fabric
- A simulation model for the HPS SDRAM memory

The HPS simulation support does not include modules implemented in the HPS, such as the ARM® Cortex™-A9 MPCore processor.

You specify simulation support files when you instantiate the HPS component in the Qsys system integration tool. When you enable a particular HPS-FPGA interface, Qsys provides the corresponding model during the generation process. Refer to "Simulation Flows" on page 29–10 for a description of the simulation flows.

For general information about instantiating the component, refer to the *Instantiating the HPS Component* chapter in volume 3 of the *Cyclone® V Device Handbook*.

The HPS simulation support enables you to develop and verify your own FPGA soft logic or intellectual property (IP) that interfaces to the HPS component.

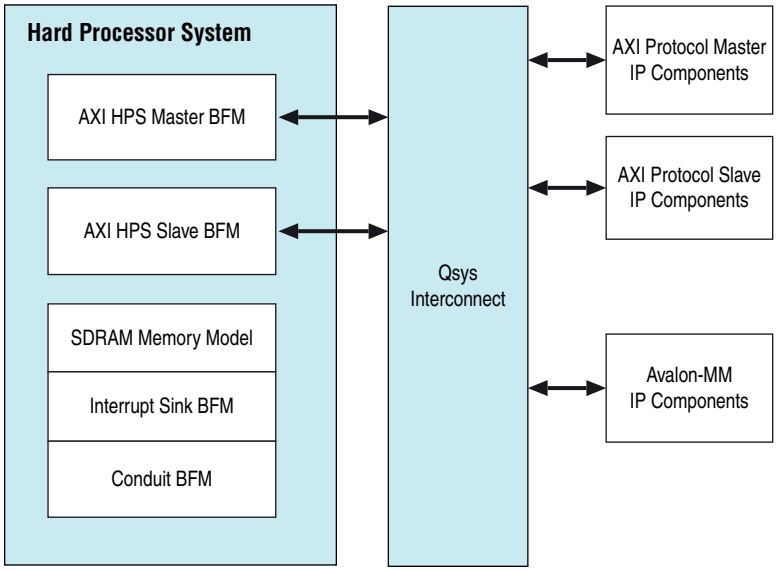
The simulation model supports the following interfaces:

- 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 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
- FPGA trace port interface

Figure 29–1 on page 29–2 shows the HPS with its BFMs.

Figure 29–1. HPS BFM Block Diagram



The HPS BFMs use standard function calls from the Altera® BFM application programming interface (API), as detailed in the remainder of this section.

For information about the BFM API, refer to the *Avalon Verification IP Suite User Guide* and the *Mentor Verification IP Altera Edition User Guide*.

HPS simulation supports only Verilog HDL or SystemVerilog simulation environments.

Clock and Reset Interfaces

For general information about clock and reset interfaces, refer to "Memory-Mapped Interfaces" in the *HPS Component Interfaces* chapter in volume 3 of the *Cyclone V Device Handbook*.

Clock Interface

Qsys 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 Qsys.

The HPS-to-FPGA trace port interface unit generates a clock output to the FPGA, named `h2f_tpiu_clock`. In simulation, the clock source BFM also represents this clock output's behavior.

Table 29–1 lists all HPS clock output interfaces with the BFM instance name.

Table 29–1. HPS Clock Output Interface Simulation Model

Interface Name	BFM Instance Name
h2f_user0_clock	h2f_user0_clock
h2f_user1_clock	h2f_user1_clock
h2f_user2_clock	h2f_user2_clock
h2f_tpiu_clock	h2f_tpiu_clock

The Altera clock source BFM application programming interface (API) applies to all the BFMs listed in Table 29–1. Your Verilog interfaces use the same API, passing in different instance names.

Qsys does not generate BFMs for FPGA-to-HPS clock input interfaces.

Reset Interface

The HPS reset request and handshake interfaces are connected to Altera conduit BFMs for simulation. Table 29–2 lists the name of each interface. You can monitor the reset request interface state changes or set the interface by using the API listed in Table 29–2.

Table 29–2. HPS Reset Input Interface Simulation Model

Interface Name	BFM Instance Name	API Function Names
f2h_cold_reset_req	f2h_cold_reset_req	get_f2h_cold_rst_req_n()
f2h_debug_reset_req	f2h_debug_reset_req	get_f2h_dbg_rst_req_n()
f2h_warm_reset_req	f2h_warm_reset_req	get_f2h_warm_rst_req_n()
h2f_warm_reset_handshake	h2f_warm_reset_handshake	set_h2f_pending_rst_req_n() get_f2h_pending_rst_ack_n()

Table 29–3 lists all HPS reset output interfaces with the BFM instance name. The Altera reset source BFM application programming interface applies to all the BFMs listed in Table 29–3.

Table 29–3. HPS Reset Output Interface Simulation Model

Interface Name	BFM Instance Name
h2f_reset	h2f_reset
h2f_cold_reset	h2f_cold_reset
h2f_debug_apb_reset	h2f_debug_apb_reset

The HPS reset output interface is connected to a reset source BFM. Qsys configures the BFM as shown in Table 29–4.

Table 29–4. Configuration of Reset Source BFM for HPS Reset Output Interface

Parameter	BFM Value (1)	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.

Note to Table 29–4:
(1) The parameter value of the instantiated BFM as configured for HPS simulation

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. Qsys configures the BFM as shown in Table 29–5. The BFM clock input is connected to `f2h_axi_clock` clock.

Table 29–5. Configuration of FPGA-to-HPS AXI Slave BFM

Parameter	Value
AXI Address Width	32
AXI Read Data Width	32, 64, 128
AXI Write Data Width	32, 64, 128
AXI ID Width	8

You control and monitor the AXI slave BFM by using the BFM API.

For more information, refer to the *Mentor Verification IP Altera Edition User Guide*. For general information about FPGA-to-HPS AXI slave interfaces, refer to "Memory-Mapped Interfaces" in the *HPS Component Interfaces* chapter in volume 3 of the *Cyclone V Device Handbook*.

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. Qsys configures the BFM as shown in Table 29–6. The BFM clock input is connected to `h2f_axi_clock` clock.

Table 29–6. Configuration of HPS-to-FPGA AXI Master BFM

Parameter	Value
AXI Address Width	30
AXI Read and Write Data Width	32, 64, 128
AXI ID Width	12

You control and monitor the AXI master BFM by using the BFM API.

For more information, refer to the *Mentor Verification IP Altera Edition User Guide*. For general information about HPS-to-FPGA AXI master interfaces, refer to "Memory-Mapped Interfaces" in the *HPS Component Interfaces* chapter in volume 3 of the *Cyclone V Device Handbook*.

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. Qsys configures the BFM as shown in Table 29–7. The BFM clock input is connected to `h2f_lw_axi_clock` clock.

Table 29–7. 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.

For more information, refer to the *Mentor Verification IP Altera Edition User Guide*. For general information about lightweight HPS-to-FPGA AXI master interfaces, refer to "Memory-Mapped Interfaces" in the *HPS Component Interfaces* chapter in volume 3 of the *Cyclone V Device Handbook*.

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.

For more information, refer to "EMI-Related HPS Features in SoC Devices" in the *Functional Description—Hard Memory Interface* chapter in volume 3 of the *External Memory Interface Handbook*.

HPS-to-FPGA MPU General-Purpose I/O Interface

The HPS-to-FPGA MPU general-purpose I/O interface is connected to an Altera conduit BFM for simulation. Table 29–8 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 in Table 29–8.

Table 29–8. HPS-to-FPGA MPU General-Purpose I/O Interface Simulation Model

Interface Name	BFM Instance Name	RTL Simulation API Function Names	Post-Fit Simulation API Function Names
h2f_mpu_gp	h2f_mpu_gp	set_h2f_mpu_gp_out() get_h2f_mpu_gp_in()	set_gp_out() get_gp_in()

HPS-to-FPGA MPU Event Interface

The HPS-to-FPGA MPU event interface is connected to an Altera conduit BFM for simulation. Table 29–9 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 in Table 29–9.

Table 29–9. HPS-to-FPGA MPU Event Interface Simulation Model

Interface Name	BFM Instance Name	RTL Simulation API Function Names (1)	Post-Fit Simulation API Function Names
h2f_mpu_events	h2f_mpu_events	get_h2f_mpu_eventi() set_h2f_mpu_evento() set_h2f_mpu_standbywfe() set_h2f_mpu_standbywfi()	get_eventi() set_evento() set_standbywfe() set_standbywfi()

Note to Table 29–9:

(1) The usage of conduit get_*() and set_*() API functions is the same as with the general Avalon conduit BFM.

FPGA-to-HPS Interrupts Interface

The FPGA-to-HPS interrupts interface is connected to an Altera Avalon interrupt sink BFM for simulation. Table 29–10 lists the name of each interface.

Table 29–10. FPGA-to-HPS Interrupts Interface Simulation Model

Interface Name	BFM Instance Name
f2h_irq0	f2h_irq0
f2h_irq1	f2h_irq1

The Altera Avalon interrupt sink BFM API applies to all the BFMs listed in Table 29–3.

HPS-to-FPGA Debug APB Interface

The HPS-to-FPGA debug APB interface is connected to an Altera conduit BFM for simulation. Table 29–11 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 in Table 29–11.

Table 29–11. 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 (STM) Hardware Event Interface

The FPGA-to-HPS STM hardware event interface is connected to an Altera conduit BFM for simulation. Table 29–12 lists the name of each interface, along with 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 in Table 29–12.

Table 29–12. 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	get_f2h_stm_hwevents()	get_stm_events()

HPS-to-FPGA Cross-Trigger Interface

The HPS-to-FPGA cross-trigger interface is connected to an Altera conduit BFM for simulation. Table 29–13 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 in Table 29–13.

Table 29–13. 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	get_h2f_cti_trig_in() set_h2f_cti_trig_in_ack() set_h2f_cti_trig_out() get_h2f_cti_trig_out_ack() set_h2f_cti_asicctl() get_h2f_cti_fpga_clk_en()	get_trig_in() set_trig_inack() set_trig_out() get_trig_outack() set_asicctl() get_clk_en()

HPS-to-FPGA Trace Port Interface

The HPS-to-FPGA trace port interface is connected to an Altera conduit BFM for simulation. Table 29–14 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 in Table 29–14.

Table 29–14. 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	get_h2f_tpiu_clk_ctl() set_h2f_tpiu_data()	get_traceclk_ctl() set_trace_data()

FPGA-to-HPS DMA Handshake Interface

The FPGA-to-HPS DMA handshake interface is connected to an Altera conduit BFM for simulation. Table 29–15 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 in Table 29–15.

Table 29–15. FPGA-to-HPS DMA Handshake Interface Simulation Model

Interface Name	BFM Instance Name	RTL Simulation API Function Names (1)	Post-Fit Simulation API Function Names
f2h_dma_req0	f2h_dma_req0	get_f2h_dma_req0_req() get_f2h_dma_req0_single() set_f2h_dma_req0_ack()	get_channel0_req() get_channel0_single() set_channel0_xx_ack()
f2h_dma_req1	f2h_dma_req1	get_f2h_dma_req1_req() get_f2h_dma_req1_single() set_f2h_dma_req1_ack()	get_channel1_req() get_channel1_single() set_channel1_xx_ack()
f2h_dma_req2	f2h_dma_req2	get_f2h_dma_req2_req() get_f2h_dma_req2_single() set_f2h_dma_req2_ack()	get_channel2_req() get_channel2_single() set_channel2_xx_ack()
f2h_dma_req3	f2h_dma_req3	get_f2h_dma_req3_req() get_f2h_dma_req3_single() set_f2h_dma_req3_ack()	get_channel3_req() get_channel3_single() set_channel3_xx_ack()
f2h_dma_req4	f2h_dma_req4	get_f2h_dma_req4_req() get_f2h_dma_req4_single() set_f2h_dma_req4_ack()	get_channel4_req() get_channel4_single() set_channel4_xx_ack()
f2h_dma_req5	f2h_dma_req5	get_f2h_dma_req5_req() get_f2h_dma_req5_single() set_f2h_dma_req5_ack()	get_channel5_req() get_channel5_single() set_channel5_xx_ack()
f2h_dma_req6	f2h_dma_req6	get_f2h_dma_req6_req() get_f2h_dma_req6_single() set_f2h_dma_req6_ack()	get_channel6_req() get_channel6_single() set_channel6_xx_ack()
f2h_dma_req7	f2h_dma_req7	get_f2h_dma_req7_req() get_f2h_dma_req7_single() set_f2h_dma_req7_ack()	get_channel7_req() get_channel7_single() set_channel7_xx_ack()

Note to Table 29–15:

(1) The usage of conduit `get_*()` and `set_*()` API functions is the same as with the general Avalon conduit BFM.

Simulation Flows

This section describes the simulation flows for an HPS-based design.

Altera provides both functional register transfer level (RTL) simulation and post-fitter gate-level simulation flows. The simulation flows involve the following major steps:

1. Instantiate the HPS component—Refer to "Specifying HPS Simulation Model in Qsys" on page 29–10.
2. Generate the system in Qsys, including the simulation model—Refer to "Generating HPS Simulation Model in Qsys" on page 29–13.
3. Run the simulation—Refer to one of the following sections:
   - "Running HPS RTL Simulation" on page 29–13
   - "Running HPS Post-Fit Simulation" on page 29–14

For general information about simulation, refer to the *Simulating Altera Designs* chapter in volume 3 of the *Quartus II Handbook*.

Specifying HPS Simulation Model in Qsys

The following steps outline how to set up the HPS component for simulation.

1. Add the HPS component from the Qsys 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 master interface in another component in the system.

   When you create your component, make sure the conduit interfaces have the correct role names, directions, and widths. Table 29–16 lists the role names, directions, and widths for all the HPS conduit interfaces.

For general information about adding the HPS component to your design, refer to the *Instantiating the HPS Component* chapter in volume 3 of the *Cyclone V Device Handbook*.

Table 29–16. HPS Conduit Interfaces (Part 1 of 3)

Role Name	Direction	Width
h2f_warm_reset_handshake		
h2f_pending_rst_req_n	Output	1
f2h_pending_rst_ack_n	Input	1
h2f_mpu_gp		
gp_in	Input	32
gp_out	Output	32

Table 29–16. HPS Conduit Interfaces (Part 2 of 3)

Role Name	Direction	Width
h2f_mpu_events		
eventi	Input	1
evento	Output	1
standbywfe	Output	2
standbywfi	Output	2
f2h_dma_req0		
req0_req	Input	1
req0_single	Input	1
req0_ack	Output	1
f2h_dma_req1		
req1_req	Input	1
req1_single	Input	1
req1_ack	Output	1
f2h_dma_req2		
req2_req	Input	1
req2_single	Input	1
req2_ack	Output	1
f2h_dma_req3		
req3_req	Input	1
req3_single	Input	1
req3_ack	Output	1
f2h_dma_req4		
req4_req	Input	1
req4_single	Input	1
req4_ack	Output	1
f2h_dma_req5		
req5_req	Input	1
req5_single	Input	1
req5_ack	Output	1
f2h_dma_req6		
req6_req	Input	1
req6_single	Input	1
req6_ack	Output	1
f2h_dma_req7		
req7_req	Input	1
req7_single	Input	1
req7_ack	Output	1

Table 29–16. HPS Conduit Interfaces (Part 3 of 3)

Role Name	Direction	Width
h2f_debug_apb		
paddr	Input	18
paddr_31	Input	1
penable	Input	1
prdata	Output	32
pready	Output	1
psel	Input	1
pslverr	Output	1
pwdata	Input	32
pwrite	Input	1
h2f_debug_apb_sideband		
pclken	Output	1
dbg_apb_disable	Output	1
f2h_stm_hw_events		
stm_hwevents	Output	28
h2f_cti		
trig_in	Output	8
trig_in_ack	Input	8
trig_out	Input	8
trig_out_ack	Output	8
asicctl	Input	8
fpga_clk_en	Output	1
h2f_tpiu		
clk_ctl	Output	1
data	Input	32

Generating HPS Simulation Model in Qsys

The following steps outline how to generate the simulation model.

1. Go to the **Generation** page in Qsys.
2. For RTL simulation, perform the following steps:
   a. Set **Create simulation model** to **Verilog**.
   b. Click **Generate**.

   HPS simulation does not support the VHDL simulation environment.

   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.
   c. Click **Generate**.

For general information about generating the HPS component, refer to the *Instantiating the HPS Component* chapter in volume 3 of the *Cyclone V Device Handbook*. For more information about Qsys simulation, refer to "Simulating a Qsys System" in the *Creating a System with Qsys* chapter in volume 1 of the *Quartus II Handbook*.

Running HPS RTL Simulation

Qsys generates scripts for various simulators that you use to complete the simulation process. Table 29–17 lists simulation tools, script names and the script directories for various vendors.

Table 29–17. Qsys-Generated Scripts for Various Simulators

Simulator	Script Name	Directory
Mentor Graphics Modelsim® Altera Edition	**msim_setup.tcl**	*<project directory>/<Qsys design name>*/**simulation/mentor**
Cadence NC-Sim	**ncsim_setup.sh**	*<project directory>/<Qsys design name>*/**simulation/cadence**
Synopsys VCS	**vcs_setup.sh**	*<project directory>/<Qsys design name>*/**simulation/synopsys/vcs**
Synopsys VCS-MX	**vcsmx_setup.sh**	*<project directory>/<Qsys design name>*/**simulation/synopsys/vcsmx**

For detailed simulation steps, refer to the *Mentor Verification IP Altera Edition User Guide*, and to the *Qsys Tutorial* chapter of the *Avalon Verification IP Suite User Guide*.

Running HPS Post-Fit Simulation

The section describes how you run HPS post-fit simulation. After successful Qsys generation, perform the following steps:

1. Add the Qsys-generated synthesis file set to your Quartus II project by performing the following steps:
   a. In the Quartus II software, click **Settings** in the Assignments menu.
   b. In the **Settings -** <*your Qsys system name*> dialog box, on the **Files** tab, browse to <*your project directory*>/<*your Qsys system name*>**/synthesis/** and select <*your Qsys system name*>**.qip**.
   c. Click **OK**.
2. You can instantiate the Qsys system that contains HPS component as your Quartus II project top-level entity, if necessary.
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**.

   For more information about EDA Netlist Writer settings, refer to *Simulation Page (Settings Dialog Box)* in Quartus II Help.
5. To create the post-fitter simulation model with Quartus II EDA Netlist Writer, in the Start menu, point to **Processing** and click **Start EDA Netlist Writer**.
6. Compile the necessary simulation files in your simulation tool. Table 29–18 lists the required libraries and files.
7. Start simulation.

Table 29–18. Post-Fit Simulation Files

Library	Directory (1)	File
Altera Verification IP Library	*<Avalon Verification IP>*/lib/	verbosity_pkg.sv avalon_mm_pkg.sv avalon_utilities_pkg.sv
Avalon Clock Source BFM	*<Avalon Verification IP>*/altera_avalon_clock_source/	altera_avalon_clock_source.sv
Avalon Reset Source BFM	*<Avalon Verification IP>*/altera_avalon_reset_source/	altera_avalon_reset_source.sv
Avalon MM Slave BFM	*<Avalon Verification IP>*/altera_avalon_mm_slave_bfm/	altera_avalon_mm_slave_bfm.sv
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/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 (2)	*<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 (3)
User testbench files	*<User project directory>*/	*.v *.sv *.vhd (3)

Notes to Table 29–18:

(1) *<ACDS install>* = Altera Complete Design 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

(2) The device simulation library is not needed with Modelsim-Altera.

(3) Mixed-language simulator is needed for Verilog HDL and VHDL mixed design

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 that consists of the HPS component.
- <*HPS*> is the HPS component instance name that you use in your Qsys system.
- <*interface*> is the instance name for 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*>**/**<*Qsys design name*>**/synthesis/submodules**.
- <*space*>—You must insert one space character after the interface instance name.
- <*BFM*> is the BFM instance name. In <*ACDS install*>**/ip/altera/hps/postfitter_simulation**, identify the SystemVerilog file corresponding to the interface type that you are using. The 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 Quartus® II software.

Document Revision History

Table 29–19 lists the revision history for this document.

Table 29–19. Document Revision History

Date	Version	Changes
November 2012	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.
June 2012	1.0	Initial release.
May 2012	0.1	Preliminary draft.

Section VIII. Appendices

This section includes the following appendices:

- Appendix A, Booting and Configuration

For information about the revision history for chapters in this section, refer to "Document Revision History" in each individual chapter.

A. Booting and Configuration

cv_5400A-1.3

This appendix describes the booting of the hard processor system (HPS) and the configuration of the FPGA portion of the Altera system-on-a-chip (SoC) FPGA device.

The HPS boot starts when a processor is released from reset (for example, on power up) and executes code in the internal boot ROM at the reset exception address. The boot process ends when the code in the boot ROM jumps to the next stage of the boot software. This next stage of boot software is referred to as the preloader. The preloader can be customized and is typically stored external to the HPS in a nonvolatile flash-based memory.

The processor can boot from the following sources:

- NAND flash memory through the NAND flash controller
- Secure Digital/MultiMediaCard (SD/MMC) flash memory through the SD/MMC flash controller
- Serial peripheral interface (SPI) and quad SPI flash memory through the quad SPI flash controller
- FPGA fabric

The HPS boot supports indirect or direct execution of the preloader depending on the boot device. With indirect execution, the boot ROM code copies the preloader from the boot device into the on-chip RAM and jumps to it. Indirect execution is used for flash memory boot sources. With direct execution, the boot ROM code executes the preloader directly from the boot device or from the FPGA fabric boot source.

Configuration of the FPGA portion of the device starts when the FPGA portion is released from the reset state (for example, on power-on). 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 non-volatile flash-based memory. The FPGA CB can obtain a configuration image from the HPS through the FPGA manager or from any of the sources supported by the Cyclone® V FPGAs family.

For more information about the memory and peripheral modules used during the booting and configuration process, refer to their respective chapters:

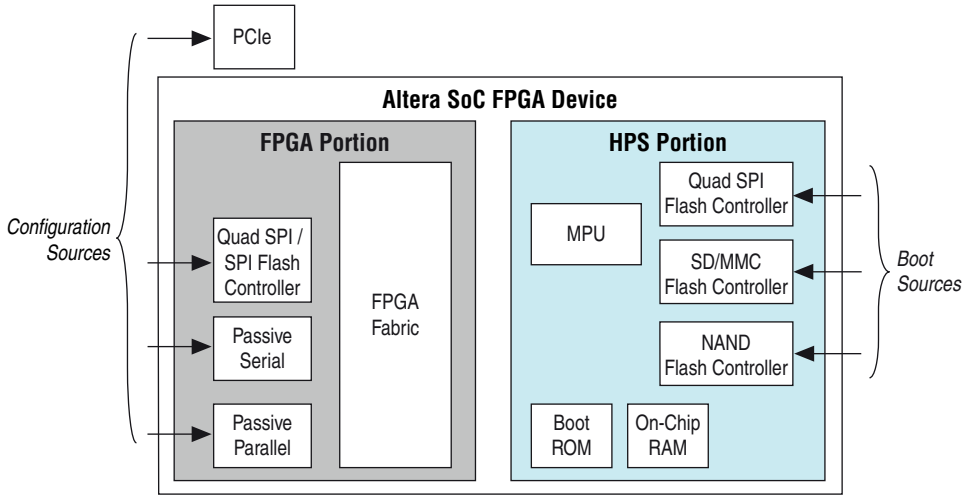
- *On-Chip Memory* chapter in volume 3 of the *Cyclone V Device Handbook*.
- *FPGA Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

The following three figures illustrate the possible HPS boot and FPGA configuration schemes. The arrows in the figures denote the data flow direction.

- Independent

  Figure A–1 shows that the FPGA configuration and HPS boot occur independently. The FPGA configuration obtains its configuration image from a non-HPS source, while the HPS boot obtains its preloader from a non-FPGA fabric source.

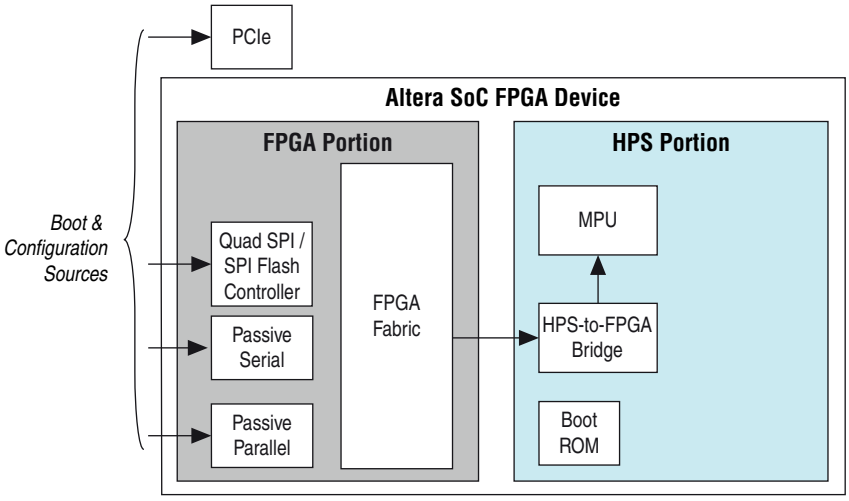
Figure A–1. Independent FPGA Configuration and HPS Booting



- FPGA first

  Figure A–2 shows that the FPGA is first configured through one of its non-HPS configuration sources and then the HPS boots from the FPGA fabric. The HPS boot waits for the FPGA fabric to be powered on and in user mode before executing. The HPS boot ROM code executes the preloader from the FPGA fabric over the HPS-to-FPGA bridge. The preloader can be obtained from the FPGA RAM or by accessing an external interface, depending on your design and implementation.

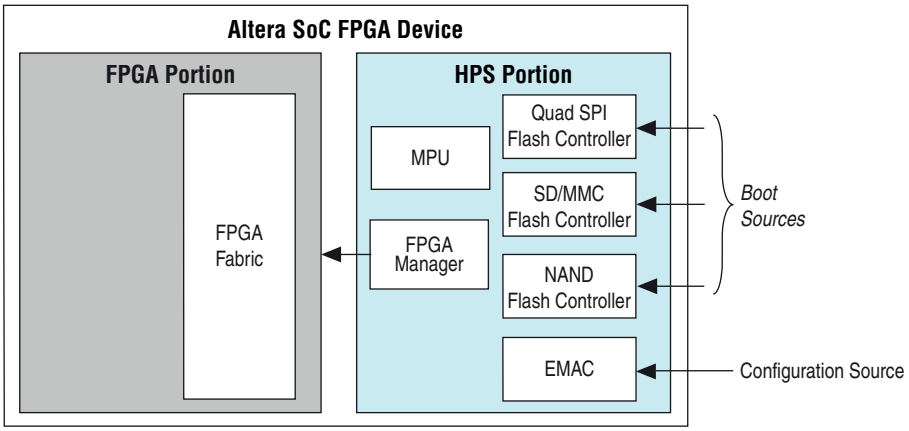
Figure A–2. FPGA Configures First



- HPS first

  Figure A–3 shows that the HPS boots first through one of its non-FPGA fabric boot sources and then software running on the HPS configures the FPGA fabric through the HPS FPGA manager. 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). The software is provided by users and the boot ROM is not involved in configuring the FPGA fabric.

Figure A–3. HPS Boots First



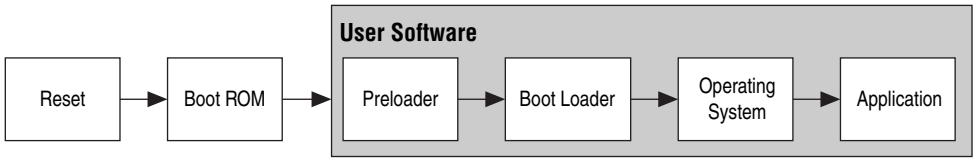
HPS Boot

Booting software on 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 stage software, called the preloader. The preloader locates, and if present, executes the next stage software. The preloader and subsequent boot stages (if present) are collectively referred to as user software.

Only the boot ROM code is located in the HPS. The user 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 boot stages.

Figure A–4 illustrates the typical boot flow. There may be more or less boot stages in the user software than shown and the roles of the boot stages may vary.

Figure A–4. Typical Boot Flow



Boot Process Overview

Reset

The boot process begins when a CPU in the MPU exits from the reset state. When a CPU exits from reset, it starts running code at the reset exception address. In normal operation, the boot ROM is mapped at the reset exception address so code starts running in the boot ROM.

It is possible to map the on-chip RAM or SDRAM at the reset exception address and run code other than the boot ROM code. However, this chapter assumes that the boot ROM maps to the reset exception address.

Boot ROM

The boot ROM contains software code that executes after a reset exits. The boot ROM code determines if it is running on CPU0 or CPU1. If running on CPU1, the boot ROM code jumps to the value in the CPU1 start address register (`cpu1startaddr`) in the boot ROM code register group (`romcodegrp`) in the system manager. If running on CPU0, the boot ROM code reads the boot select (BSEL) and clock select (CSEL) values from the `bsel` and `csel` fields of the boot information register (`bootinfo`) in the system manager to determine the boot source and to set up the clock manager. The `bsel` and `csel` field values come from the `BSEL` and `CSEL` pins. The system manager samples the values on these pins when coming out of a reset.

Table A–1 lists the `bsel` field values for each flash memory device selection.

Table A–1. bsel Field Values and Flash Device Selection

bsel Field Value	Flash Device
0x0	Reserved
0x1	FPGA (HPS-to-FPGA bridge)
0x2	1.8 V NAND flash memory
0x3	3.0 V NAND flash memory
0x4	1.8 V SD/MMC flash memory with external transceiver
0x5	3.0 V SD/MMC flash memory with internal transceiver
0x6	1.8 V SPI or quad SPI flash memory
0x7	3.0 V SPI or quad SPI flash memory

For indirect execution from flash memory boot sources, the boot ROM code loads the preloader image from the flash device to the on-chip RAM and passes software control to the preloader in the on-chip RAM. For direct execution from the FPGA fabric boot source, the boot ROM code waits until the FPGA portion of the device is in user mode and then passes software control to the preloader in the FPGA RAM.

Preloader

The function of the preloader is user-defined. However, typical functions include initializing the SDRAM interface and configuring the HPS I/O pins. Initializing the SDRAM allows the preloader to load the next stage of the boot software (that might not fit in the 60 kilobytes (KB) available in the on-chip RAM). A typical next software stage is the open source boot loader, U-boot.

The preloader is allowed to load the next stage boot software 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.

Boot Loader

The boot loader loads the operating system and passes software control to the operating system.

Boot ROM

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

- Sets up the level 4 (l4) watchdog 0 timer
- Configures the main PLL and peripheral PLL based on the CSEL value
- Configures I/O elements and pin multiplexing based on the BSEL value
- Initializes the flash controller to default settings

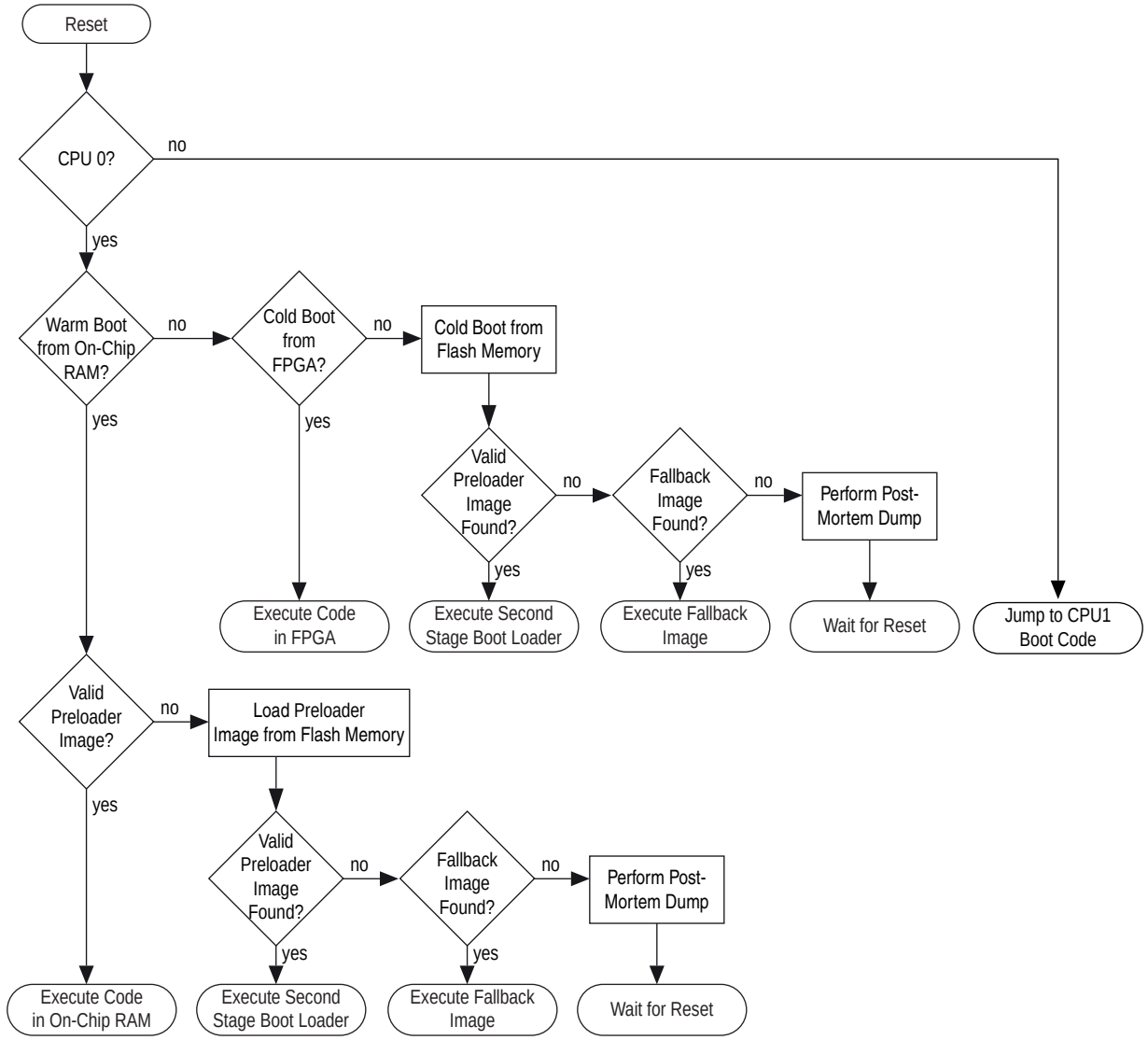
While executing, the boot ROM code uses the top 4 KB of the on-chip RAM as data workspace. This area is reserved for the boot ROM code after a reset until the boot ROM code passes software control to preloader. This limits the maximum size of the preloader for indirect execution to 60 KB.

Boot ROM Flow

This section describes the software flow from reset until the boot ROM code passes software control to the preloader.

Figure A–5 illustrates the boot ROM flow. 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–5. Boot ROM Flow



During a cold boot from the FPGA portion of the device, the boot ROM code waits until the FPGA is ready and then attempts direct execution at address 0x0 across the HPS-to-FPGA bridge. For example, the boot software could be provided by initialized on-chip RAM in the FPGA portion of the device at address 0x0. During a cold boot from flash memory, the boot ROM code attempts to load the first preloader image from flash memory to on-chip RAM and pass control to the preloader. If the image is invalid, the boot ROM code attempts to load up to three subsequent images from flash memory. If there is still no valid image found after the subsequent loads, the boot ROM code checks the FPGA portion of the device for a fallback image.

During a warm boot from on-chip RAM, the boot ROM code reads the preloader state register (`initswstate`) in the `romcodegrp` group in the system manager to determine if there is a valid preloader image in the on-chip RAM. If a valid preloader image is found in the on-chip RAM, the boot ROM code skips loading a preloader image from flash memory and instead passes control to the preloader residing in the on-chip RAM.

If a valid preloader image cannot be found in the on-chip RAM, 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.

Loading the Preloader

For indirect execution, the boot ROM code loads the preloader 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. Figure A–6 shows the preloader header.

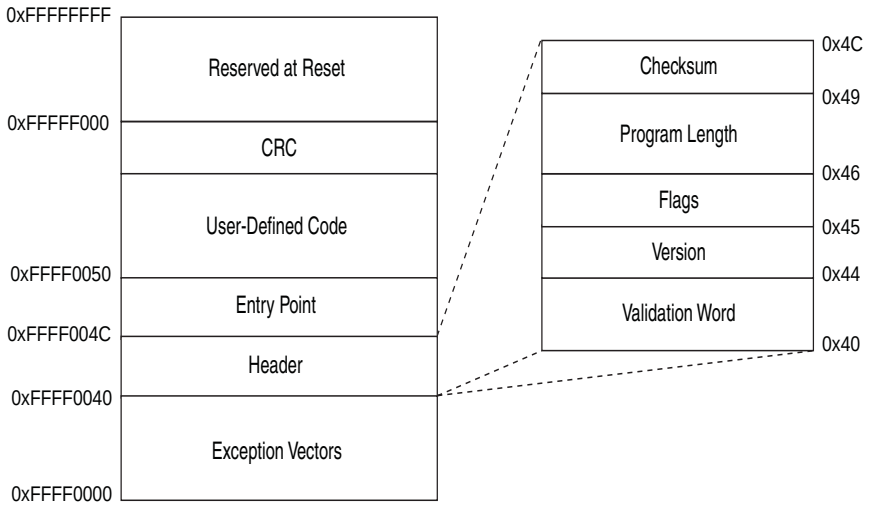
The boot ROM code checks the header for the following information:

- Validation word—validates the preloader image. The validation word has a fixed value of 0x31305341.
- Version—indicates the preloader version.
- Flags—unused
- Program length—the total length of the image (in bytes) 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.

Figure A–6 shows the preloader image layout in the on-chip RAM after being loaded from the boot ROM.

Figure A–6. Preloader Image Layout *(1)*



Note to Figure A–6:

(1) Addresses are not represented to scale.

Exception vectors—Exception vectors are located at the start of the on-chip RAM. Typically, the preloader 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 image before passing control to the preloader.

Entry point—contains the preloader image address. 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.

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 "HPS State on Entry to the Preloader" on page A–10.

The shared memory contains the following information:

- Common—contains non-flash-specific settings used by the boot ROM code.
- Saved hardware register—contains hardware register values that the boot ROM code saves before the registers are modified by the boot ROM code.
- Flash device-specific—contains flash device-specific settings used by the boot ROM code that the preloader may use to continue using the flash device without reinitialization.

Table A–2 lists the contents of the shared memory block.

Table A–2. Shared Memory Block (Part 1 of 2)

Information Type	Content	Description
Common	Flash image	Indicates which preloader image (0-3) the boot ROM code loaded from the flash device. A nonzero value indicates that there was an error loading image 0 and the boot ROM code loaded another image.
	CSEL value used	Indicates the CSEL value used by the boot ROM code. Typically, this value is the value read from the `csel` field of the `bootinfo` register in the `romcodegrp` group in the system manager. However, if the PLL fails to lock, the boot ROM code ignores the `csel` field and uses zero to indicate PLL bypass mode.
	BSEL value used	Indicates the BSEL value used by the boot ROM code. Typically, this value is the value read from the `bsel` field of the `bootinfo` register in the `romcodegrp` group in the system manager.
	Last page	Indicates the address of the last page read from the flash device.
	Page size	Indicates the page size used by the flash device. ■ For NAND flash memory, the boot ROM code reads this value from the NAND flash controller. ■ For SPI and quad SPI flash memory, the boot ROM code configures the page size through the quad SPI flash controller. ■ For SD/MMC flash memory, the boot ROM code configures the page size through the SD/MMC flash controller.
	Flash device type	Indicates the flash device used by the boot ROM code.
	Step complete	Track the completion state of up to 64 individual major steps during the boot process. The 64 bit value has one bit set for each major step completed in the boot process.
	CPU0 and CPU1 crash data	Indicates the CPU0 and CPU1 crash data. These values are pointers to where the boot ROM code saves the crash dump in an event of a crash.
Saved hardware registers	Status register (`stat`) of the reset manager	Contains reset source and event timeout information.
	Control (`ctrl`) register in the `romcodegrp` group in the system manager	Contains information used to control the boot ROM code.
	`initswstate` register in the `romcodegrp` group in the system manager	Contains the magic value 0x49535756 when the preloader has reached a valid state.

Table A–2. Shared Memory Block (Part 2 of 2)

Information Type	Content	Description
Flash device-specific (SD/MMC)	is_sd_card	Indicates the card type. 1 indicates SD card; 0 indicates MMC.
	is_sector_mode	Indicates the addressing mode of card. 1 indicates sector addressing mode; 0 indicates byte addressing mode.
	rca	Contains the relative card address, which is used for host-card communication during card identification
	partition_start_sector	Indicates the partition start offset in unit sectors. 0 indicates raw mode.
	partition_size	Size of the partition in unit sectors. 0 indicates raw mode.

L4 Watchdog 0 Timer

The L4 watchdog 0 timer is reserved for boot ROM use. If a watchdog reset happens before software control passes to the preloader, boot ROM code attempts to load the last valid preloader image, identified by the `initswlastld` register in the `romcodegrp` group in the system manager.

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 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.
- r3—unused and set to 0x0.
- r4—unused and set to 0x0.

All other MPCore registers are undefined.

☞ 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.

Preloader

The preloader typically performs the following actions:

- Initialize the SDRAM interface.
- Configure the `remap` register of the L3 (NIC-301) GPV registers (`l3regs`) to map the on-chip RAM to address 0x0 so that exceptions are handled by the preloader.

  ☞ The on-chip RAM is also accessible at address 0xFFFF0000. The address 0x0 is an alias.

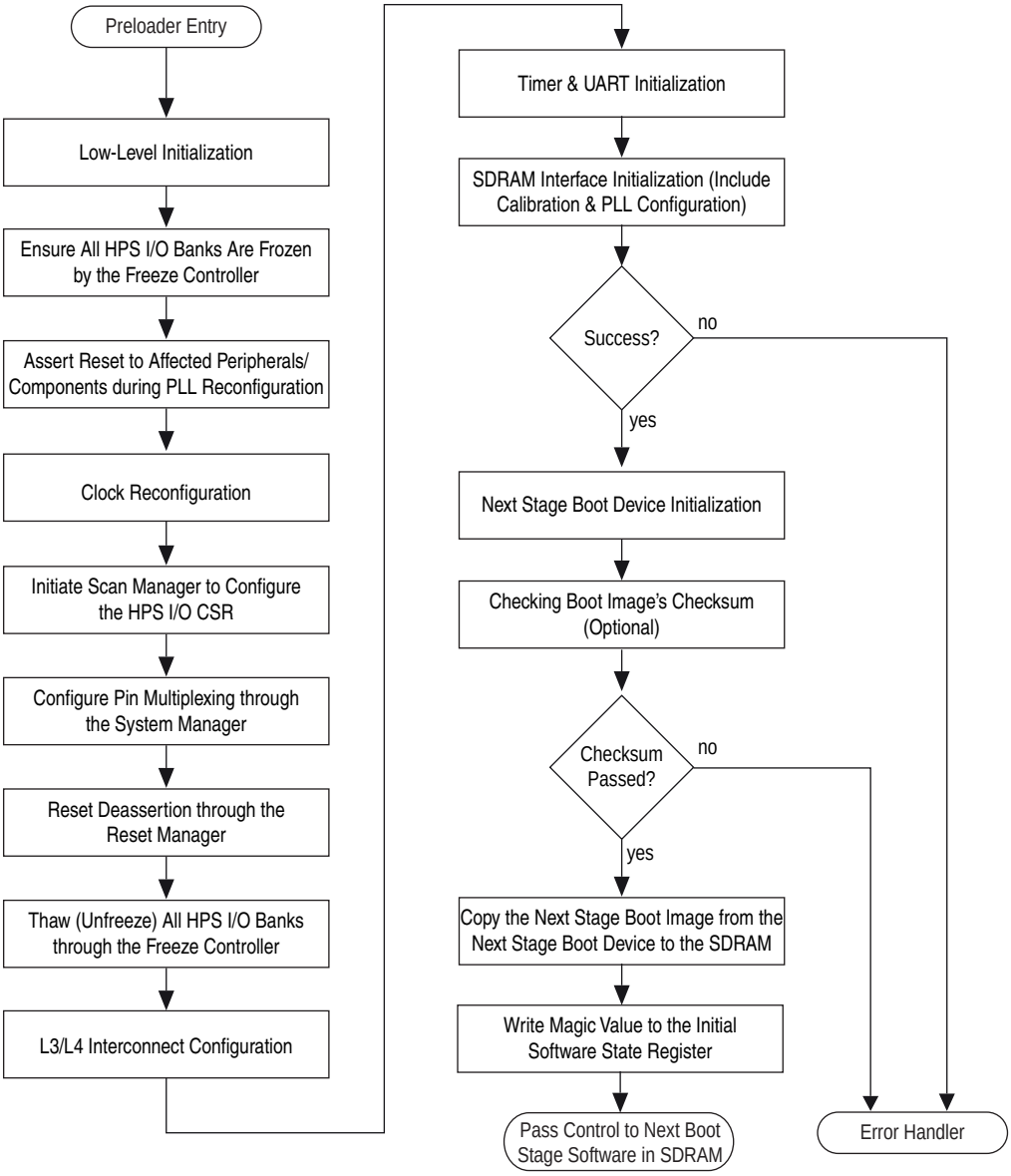
- Configure the HPS I/O through the scan manager.
- Configure pin multiplexing through the system manager.
- Configure HPS clocks through the clock manager.
- Initialize the flash controller (NAND, SD/MMC, or quad SPI) that contains the next stage boot software.
- Load the next stage boot software into the SDRAM and pass control to it.

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–7 shows a typical preloader boot flow.

Figure A–7. Typical Preloader Boot 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.

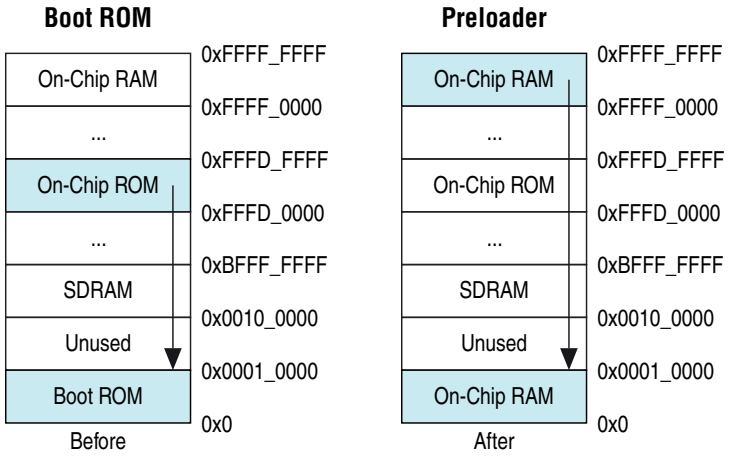
Upon entering the preloader, the L4 watchdog 0 timer is active. The preloader can either disable, reconfigure, or leave the watchdog timer unchanged. Once enabled after reset, the watchdog timer cannot be disabled, only paused.

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 will use the exception handlers in the preloader image.

Figure A–8 shows the memory map before and after remap.

Figure A–8. Remapping the On-Chip RAM *(1)*



Note to Figure A–8:

(1) Addresses are not represented to scale.

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.

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 (such as 0x49535756) to the preloader `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. When a warm reset occurs, the boot ROM code can check the `initswstate` register for the magic value to determine if it needs to reload the preloader image into the on-chip RAM.

Flash Memory Devices

The flash memory devices available for HPS boot are the NAND, SD/MMC, SPI, and quad SPI. The flash device can store the following kinds of files for booting purposes:

- Preloader binary file (up to four copies)
- Boot loader binary file
- Operating system binary file
- Application file
- FPGA programming files

The preloader file must be stored in a partition with no file system.

NAND Flash Devices

Figure A–9 shows the NAND flash image layout. The preloader image is located at offsets which are multiples of the block size. If the image size is less than 64 KB, only one block size is used. Since a block is the smallest area used for erase operation, any update to a particular image does not affect other images.

Figure A–9. NAND Flash Image Layout

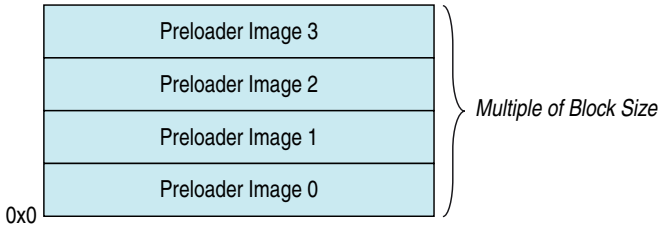


Table A–3 lists the NAND flash driver features supported in the boot ROM code.

Table A–3. 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) devices with integrated error correction code (ECC).
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	x8 only
Page size	512 bytes, 2 KB, 4 KB, or 8 KB.
Page per block	8, 16, 32, or 128
ECC	512-bytes with 8-bit correction

Table A–4 lists the CSEL pin settings for the quad SPI controller.

Table A–4. NAND Controller CSEL Pin Settings

Setting	CSEL Pin			
	0	1	2	3
osc1_clk (EOSC1 pin) range	10–50 MHz	10–12.5 MHz	12.5–25 MHz	25–50 MHz
nand_x_clk /25 device frequency	osc1_clk/25, 2 MHz max	osc1_clk*20/25, 9.6 MHz max	osc1_clk*10/25, 9.6 MHz max	osc1_clk*5/25, 9.6 MHz max
nand_x_clk controller clock	osc1_clk, 50 MHz max	osc1_clk*20, 240 MHz max	osc1_clk*10, 240 MHz max	osc1_clk*5, 240 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

For more information about the NAND flash memory, refer to the *NAND Flash Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

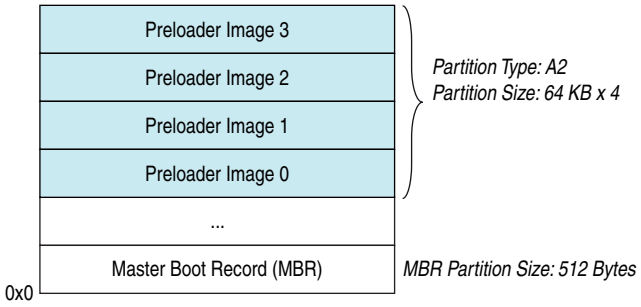
SD/MMC Flash Devices

Figure A–10 shows the SD/MMC flash image layout. The master boot record (MBR) is located at the first 512 bytes of the memory. The MBR contains information of partitions (address and size of partition). The preloader image is always stored in partition A2. Partition A2 is a custom raw partition with no file system.

The start address of each image is based on the following formula:

Start address = start address + (<*n*> * 64 K), where <*n*> is the image number

Figure A–10. 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 0
  - Start address = 0

MBR

The MBR contains the partition table, which is always located on 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–5 lists the MBR structure.

Table A–5. MBR Structure

Offset	Size (Byte)	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-bytes 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 file system, so the boot images are located in partition A2 at fixed locations. Table A–6 lists the partition entry.

Table A–6. Partition Entry

Offset	Size (Byte)	Description
0x0	1	Boot indicator. 0x80 indicates that is it bootable.
0x1	3	Starting CHS value
0x4	1	Partition type
0x5	3	Ending CHS value
0x8	4	LBA of first section 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. Table A–7 lists the default settings of the SD/MMC controller.

Table A–7. 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
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
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*16=32)
	Data transfer mode	Bypass	The clock divider register (clkdiv)= 0x00

Table A–8 lists the CSEL pin settings for the SD/MMC controller.

Table A–8. SD/MMC Controller CSEL Pin Settings

Setting		CSEL Pin			
		0	1	2	3
osc1_clk (EOSC1 pin) range		10–50 MHz	10–12.5 MHz	12.5–25 MHz	25–50 MHz
ID mode	sdmmc_cclk_out device clock	osc1_clk/128, 391 KHz max	osc1_clk/32, 391 KHz max	osc1_clk/64, 391 KHz max	osc1_clk/128, 391 KHz max
	Controller baud rate divisor	32	32	32	32
Data transfer mode	sdmmc_cclk_out device clock	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)
sdmmc_clk controller clock:		osc1_clk, 50 MHz max	osc1_clk, 50 MHz max	osc1_clk, 50 MHz max	osc1_clk*2, 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

For more information about the SD/MMC, refer to the *SD/MMC Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

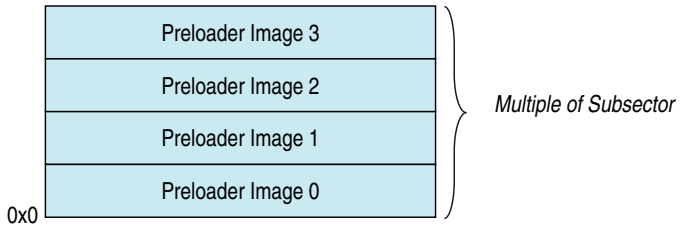
SPI and Quad SPI Flash Devices

Figure A–11 shows the SPI and quad SPI flash image layout. The preloader image is always located at offsets which are multiples of a subsector size. If the image is less than 64 KB, only one subsector is used. If subsector erase is not supported, sector erase is supported. Since a subsector is the smallest area used for erase operation, any update to a particular image does not affect other images.

The first image is located at offset 0 and followed by subsequent images. The start address of each image is based on the following formula:

Start address = (N * 64K), where N is the image number

Figure A–11. SPI and 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. Table A–9 lists the default settings of the quad SPI controller.

Table A–9. 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.	Normal read	The read opcode in non-XIP mode field (`rdopcode`) in the device read instruction register (`devrd`) = 0x3.
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.
Delay in master reference clocks for the length that the master mode chip select outputs are deasserted between words when the clock phase is zero.	200 ns	The clock delay for chip select deassert field (`nss`) in the quad SPI device delay register (`delay`). Refer to `delay[31:24]` in Table A–11 on page A–19.
Delay in master reference clocks 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.	0 ns	The clock delay for chip select deactivation field (`btwn`) in the `delay` register = 0x0.
Delay in master reference clocks 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.	20 ns	The clock delay for last transaction bit field (`after`) in the `delay` register. Refer to `delay[15:8]` in Table A–11 on page A–19.
Added delay in master reference clocks between setting `qspi_n_ss_out` low and first bit transfer.	20 ns	The clock delay with `qspi_n_ss_out` field (`init`) in the `delay` register. Refer to `delay[7:0]` in Table A–11 on page A–19.
Number of address bytes.	3 bytes	The number of address bytes field (`numaddrbytes`) of the device size register (`devsz`) = 2.

Table A–10 lists the CSEL pin settings for the quad SPI controller.

Table A–10. Quad SPI Controller CSEL Pin Settings

Setting	CSEL Pin			
	0	1	2	3
osc1_clk (EOSC1 pin) range	10–50 MHz	20–50 MHz	25–50 MHz	10–25 MHz
sclk_out device clock	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
qspi_clk controller clock	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

The delay register in the quad SPI controller registers (qspiregs) configures relative delay into the generation of the master output signals.

The SPI or quad SPI flash memory needs to meet the following timing requirements:

- T_{SLCH} (delay[7:0]): 20 ns
- T_{CHSH} (delay[15:8]): 20 ns
- T_{SHSL} (delay[31:24]): 200 ns

Table A–11 lists the delay register configuration with CSEL pin settings.

Table A–11. SPI and Quad SPI Flash Delay Configuration

CSEL Pin	T_{ref_clk} (ns)	T_{sclk_out} (ns)	Device Delay Register		
			delay[7:0]	delay[15:8]	delay[31:24]
0	20	80	0	0	6
1	10	40	0	0	16
2–3	5	20	0	0	36

The formula to calculate the delay is $(T_{delay}-T_{sclk\_out}) / T_{ref\_clk}$.

For more information about the quad SPI flash memory, refer to the *Quad SPI Flash Controller* chapter in volume 3 of the *Cyclone V Device Handbook*.

FPGA Configuration

You can configure the FPGA portion of the SoC device with non-HPS sources or by utilizing the HPS. Software executing on the HPS configures the FPGA by writing 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.

For more information about configuring the FPGA through the HPS FPGA manager, refer to the *FPGA Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

For more information about configuring the FPGA in general, refer to the *Configuration, Design Security, and Remote System Upgrades in Cyclone V Devices* chapter in volume 1 of the *Cyclone V Device Handbook*.

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:

1. Set the `cdratio` and `cfgwdth` bits of the `ctrl` register in the FPGA manager registers (`fpgamgrregs`) to match the characteristics of the configuration image. These settings are dependant 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 state of the `nSTATUS` bits (`ns`) in the configuration monitor registers (`fpgamgrregs.mon`).
9. Set the `axicfgen` 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` registers to monitor the `CONF_DONE` (`cd`) and `nSTATUS` (`ns`) bits.
    a. `CONF_DONE` = 1 and `nSTATUS` = 1 indicates successful configuration.
    b. `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.
12. Set the `axicfgen` bit of the `ctrl` register to 0 to disable configuration data on AXI slave.
13. Send the `DCLKs` required by the FPGA to enter the initialization phase.
    a. If `DCLK` is unused, write a value of 4 to the `DCLK` count register (`dclkcnt`).
    b. If `DCLK` is used, write a value of 20,480 (0x5000) to the `dclkcnt` register.

14. Poll the dcntdone bit of the DCLK status register (dclkstat) until it changes to 1, which indicates that all the DCLKs have been sent.
15. Write a 1 to the dcntdone bit of the DCLK status register to clear the completed status flag.
16. Read the mode bit of the stat register to wait for the FPGA to enter user mode.
17. Set the en bit of the ctrl register to 0 to allow the external pins to drive the configuration input signals.

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.

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:

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 cfgwdth 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.
   a. If PR_READY=1, continue to step 7.
   b. If PR_READY is 0, then go back and repeat step 5. Note that a minimum of 16 DCLK pulses are required.
7. Write a value of 3 to the dclkcnt register to generate DCLK pulses for three clock cycles.
8. Set the axicfgen 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.

Table A–12. Partial Reconfiguration Status

Bits				Partial Reconfiguration Status
PR_DONE	PR_READY	PR_ERROR	CRC_ERROR	
1	0	0	0	Completed
0	0	1	0	Completed with error
0	0	0	1	Completed with a SEU event CRC error

11. Set the axicfgen 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.

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 A–20.

For more information about the configuration modes and pin settings, refer to the *FPGA Manager* chapter in volume 3 of the *Cyclone V Device Handbook*.

Document Revision History

Table A–13 shows the revision history for this document.

Table A–13. Document Revision History (Part 1 of 2)

Date	Version	Changes
November 2012	1.3	■ Expanded shared memory block table. ■ Added CSEL tables. ■ Additional minor updates.
June 2012	1.2	Updated the HPS boot and FPGA configuration sections.

Table A–13. Document Revision History (Part 2 of 2)

Date	Version	Changes
May 2012	1.1	■ Updated the HPS boot section. ■ Added information about the flash devices used for HPS boot. ■ Added information about the FPGA configuration mode.
January 2012	1.0	Initial release.

This chapter provides additional information about the document and Altera.

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Nontechnical support (general)	Email	nacomp@altera.com
(software licensing)	Email	authorization@altera.com

Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, *<file name>* and *<project name>***.pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections in a document and titles of Quartus II Help topics. For example, "Typographic Conventions."

Visual Cue	Meaning
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix n denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
↵	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	The question mark directs you to a software help system with related information.
	The feet direct you to another document or website with related information.
	The multimedia icon directs you to a related multimedia presentation.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.