Embedded Peripherals IP

User Guide

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

UG-01085-10.0.0

Copyright © 2010 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

I.S. EN ISO 9001

Contents

Chapter Revision Dates xix

Section I. Off-Chip Interface Peripherals

Chapter 1. SDRAM Controller Core

Core Overview 1-1
Functional Description 1-2
Avalon-MM Interface 1-2
Off-Chip SDRAM Interface 1-3
Signal Timing and Electrical Characteristics 1-3
Synchronizing Clock and Data Signals 1-3
Clock Enable (CKE) Not Supported 1-3
Sharing Pins with Other Avalon-MM Tri-State Devices 1-3
Board Layout and Pinout Considerations 1-4
Performance Considerations 1-4
Open Row Management 1-4
Sharing Data and Address Pins 1-4
Hardware Design and Target Device 1-5
Device Support 1-5
Instantiating the Core in SOPC Builder 1-5
Memory Profile Page 1-6
Timing Page 1-7
Hardware Simulation Considerations 1-7
SDRAM Controller Simulation Model 1-8
SDRAM Memory Model 1-8
Using the Generic Memory Model 1-8
Using the SDRAM Manufacturer's Memory Model 1-8
Example Configurations 1-8
Software Programming Model 1-10
Clock, PLL and Timing Considerations 1-10
Factors Affecting SDRAM Timing 1-11
Symptoms of an Untuned PLL 1-11
Estimating the Valid Signal Window 1-11
Example Calculation 1-13
Referenced Documents 1-15
Document Revision History 1-16

Chapter 2. CompactFlash Core

Core Overview 2-1
Functional Description 2-1
Instantiating the Core in SOPC Builder 2-2
Required Connections 2-2
Device Support 2-3
Software Programming Model 2-3
HAL System Library Support 2-3
Software Files 2-4
Register Maps 2-4
Ide Registers 2-4

Ctl Registers 2-4
Document Revision History 2-5

Chapter 3. Common Flash Interface Controller Core
Core Overview 3-1
Functional Description 3-2
Device and Tools Support 3-2
Instantiating the Core in SOPC Builder 3-2
Attributes Page 3-3
Presets Settings 3-3
Size Settings 3-3
Timing Page 3-3
Software Programming Model 3-4
HAL System Library Support 3-4
Limitations 3-4
Software Files 3-4
Referenced Documents 3-5
Document Revision History 3-5

Chapter 4. EPCS Serial Flash Controller Core
Core Overview 4-1
Functional Description 4-2
Avalon-MM Slave Interface and Registers 4-3
Device and Tools Support 4-4
Instantiating the Core in SOPC Builder 4-4
Software Programming Model 4-4
HAL System Library Support 4-4
Software Files 4-5
Referenced Documents 4-5
Document Revision History 4-5

Chapter 5. JTAG UART Core
Core Overview 5-1
Functional Description 5-2
Avalon Slave Interface and Registers 5-2
Read and Write FIFOs 5-2
JTAG Interface 5-3
Host-Target Connection 5-3
Device and Tools Support 5-4
Instantiating the Core in SOPC Builder 5-4
Configuration Page 5-4
Write FIFO Settings 5-4
Read FIFO Settings 5-5
Simulation Settings 5-5
Simulated Input Character Stream 5-5
Prepare Interactive Windows 5-5
Hardware Simulation Considerations 5-6
Software Programming Model 5-6
HAL System Library Support 5-6
Driver Options: Fast vs. Small Implementations 5-8
ioctl() Operations 5-8
Software Files 5-9
Accessing the JTAG UART Core via a Host PC 5-9

Register Map 5-9
Data Register 5-10
Control Register 5-10
Interrupt Behavior 5-11
Referenced Documents 5-12
Document Revision History 5-12

Chapter 6. UART Core

Core Overview 6-1
Functional Description 6-1
Avalon-MM Slave Interface and Registers 6-2
RS-232 Interface 6-2
Transmitter Logic 6-2
Receiver Logic 6-3
Baud Rate Generation 6-3
Device Support 6-3
Instantiating the Core in SOPC Builder 6-3
Configuration Settings 6-4
Baud Rate Options 6-4
Data Bits, Stop Bits, Parity 6-5
Synchronizer Stages 6-5
Flow Control 6-5
Streaming Data (DMA) Control 6-6
Simulation Settings 6-6
Simulated RXD-Input Character Stream 6-7
Prepare Interactive Windows 6-7
Simulated Transmitter Baud Rate 6-7
Simulation Considerations 6-7
Software Programming Model 6-8
HAL System Library Support 6-8
Driver Options: Fast Versus Small Implementations 6-9
ioctl() Operations 6-10
Limitations 6-10
Software Files 6-10
Register Map 6-11
rxdata Register 6-11
txdata Register 6-12
status Register 6-12
control Register 6-14
divisor Register (Optional) 6-14
endofpacket Register (Optional) 6-15
Interrupt Behavior 6-15
Referenced Documents 6-15
Document Revision History 6-16

Chapter 7. SPI Core

Core Overview 7-1
Functional Description 7-1
Example Configurations 7-2
Transmitter Logic 7-3
Receiver Logic 7-3
Master and Slave Modes 7-3
Master Mode Operation 7-3

Slave Mode Operation . . . 7-4
Multi-Slave Environments . . . 7-5
Avalon-MM Interface . . . 7-5
Instantiating the SPI Core in SOPC Builder . . . 7-5
Master/Slave Settings . . . 7-5
Number of Select (SS_n) Signals . . . 7-5
SPI Clock (sclk) Rate . . . 7-6
Specify Delay . . . 7-6
Data Register Settings . . . 7-6
Timing Settings . . . 7-7
Device Support . . . 7-8
Software Programming Model . . . 7-8
Hardware Access Routines . . . 7-8
alt_avalon_spi_command() . . . 7-9
Software Files . . . 7-9
Register Map . . . 7-9
rxdata Register . . . 7-10
txdata Register . . . 7-10
status Register . . . 7-11
control Register . . . 7-12
slaveselect Register . . . 7-12
Referenced Documents . . . 7-12
Document Revision History . . . 7-13

Chapter 8. Optrex 16207 LCD Controller Core

Core Overview . . . 8-1
Functional Description . . . 8-1
Device and Tools Support . . . 8-2
Instantiating the Core in SOPC Builder . . . 8-2
Software Programming Model . . . 8-2
HAL System Library Support . . . 8-2
Displaying Characters on the LCD . . . 8-3
Software Files . . . 8-3
Register Map . . . 8-4
Interrupt Behavior . . . 8-4
Referenced Documents . . . 8-4
Document Revision History . . . 8-4

Chapter 9. PIO Core

Core Overview . . . 9-1
Functional Description . . . 9-1
Data Input and Output . . . 9-2
Edge Capture . . . 9-2
IRQ Generation . . . 9-3
Example Configurations . . . 9-3
Avalon-MM Interface . . . 9-3
Instantiating the PIO Core in SOPC Builder . . . 9-4
Basic Settings . . . 9-4
Width . . . 9-4
Direction . . . 9-4
Output Port Reset Value . . . 9-4
Output Register . . . 9-4
Input Options . . . 9-4

Edge Capture Register 9-4
Interrupt 9-5
Simulation 9-5
Device Support 9-5
Software Programming Model 9-5
Software Files 9-5
Register Map 9-6
data Register 9-6
direction Register 9-6
interruptmask Register 9-7
edgecapture Register 9-7
outset and outclear Registers 9-7
Interrupt Behavior 9-7
Software Files 9-7
Document Revision History 9-8

Chapter 10. Avalon-ST Serial Peripheral Interface Core

Core Overview 10-1
Functional Description 10-1
Interfaces 10-2
Operation 10-2
Timing 10-3
Limitations 10-3
Instantiating the Core in SOPC Builder 10-3
Device Support 10-3
Referenced Documents 10-4
Document Revision History 10-4

Chapter 11. PCI Lite Core

Core Overview 11-1
Performance and Resource Utilization 11-1
Functional Description 11-2
PCI-Avalon Bridge Blocks 11-2
Avalon-MM Ports 11-3
Master and Target Performance 11-5
Master Performance 11-5
Target Performance 11-5
PCI-to-Avalon Address Translation 11-6
Avalon-to-PCI Address Translation 11-6
Avalon-To-PCI Read and Write Operation 11-8
Avalon-to-PCI Write Requests 11-9
Avalon-to-PCI Read Requests 11-9
Ordering of Requests 11-10
PCI Interrupt 11-10
Instantiating the Core in SOPC Builder 11-11
PCI Timing Constraint Files 11-12
Additional Tcl Option 11-13
Device Support 11-14
Simulation Considerations 11-14
Features 11-14
Master Transactor (mstr_tranx) 11-14
TASKS Sections 11-14
INITIALIZATION Section 11-15

USER COMMANDS Section 11-15
Simulation Flow 11-15
Referenced Documents 11-17
Document Revision History 11-17

Chapter 12. Cyclone III Remote Update Controller Core
Core Overview 12-1
Functional Description 12-1
Avalon-MM Slave Interface and Registers 12-2
Device Support 12-2
Instantiating the Core in SOPC Builder 12-2
Software Programming Model 12-3
Setting the Configuration Offset 12-3
Shifting the Configuration Offset Value 12-3
Setting up the Watchdog Timer 12-3
Triggering a Reconfiguration 12-4
Code Example 12-5
Related Documentation 12-6
Document Revision History 12-6

Chapter 13. MDIO Core
Core Overview 13-1
Functional Description 13-1
MDIO Frame Format (Clause 45) 13-2
MDIO Clock Generation 13-3
Interfaces 13-3
Operation 13-4
Write 13-4
Read 13-4
Instantiating the Core in SOPC Builder 13-4
Register Map 13-5
Device Support 13-5
Referenced Documents 13-5
Document Revision History 13-5

Section II. On-Chip Storage Peripherals

Chapter 14. Avalon-ST Single-Clock and Dual-Clock FIFO Cores
Core Overview 14-1
Functional Description 14-1
Interfaces 14-2
Avalon-ST Data Interface 14-2
Avalon-MM Control and Status Register Interface 14-2
Avalon-ST Status Interface 14-3
Operating Modes 14-3
Fill Level 14-3
Thresholds 14-4
Instantiating the Core in SOPC Builder 14-4
Register Description 14-5
Device Support 14-6
Referenced Documents 14-6
Document Revision History 14-6

Chapter 15. On-Chip FIFO Memory Core

Core Overview . . . 15-1
Functional Description . . . 15-1
Avalon-MM Write Slave to Avalon-MM Read Slave . . . 15-2
Avalon-ST Sink to Avalon-ST Source . . . 15-2
Avalon-MM Write Slave to Avalon-ST Source . . . 15-3
Avalon-ST Sink to Avalon-MM Read Slave . . . 15-5
Status Interface . . . 15-6
Clocking Modes . . . 15-6
Device Support . . . 15-6
Instantiating the Core in SOPC Builder . . . 15-6
FIFO Settings . . . 15-6
Depth . . . 15-6
Clock Settings . . . 15-6
Status Port . . . 15-6
FIFO Implementation . . . 15-7
Interface Parameters . . . 15-7
Input . . . 15-7
Output . . . 15-7
Allow Backpressure . . . 15-7
Avalon-MM Port Settings . . . 15-7
Avalon-ST Port Settings . . . 15-7
Software Programming Model . . . 15-8
HAL System Library Support . . . 15-8
Software Files . . . 15-8
Programming with the On-Chip FIFO Memory . . . 15-8
Software Control . . . 15-9
Software Example . . . 15-12
On-Chip FIFO Memory API . . . 15-13
altera_avalon_fifo_init() . . . 15-13
altera_avalon_fifo_read_status() . . . 15-13
altera_avalon_fifo_read_ienable() . . . 15-14
altera_avalon_fifo_read_almostfull() . . . 15-14
altera_avalon_fifo_read_almostempty() . . . 15-14
altera_avalon_fifo_read_event() . . . 15-14
altera_avalon_fifo_read_level() . . . 15-15
altera_avalon_fifo_clear_event() . . . 15-15
altera_avalon_fifo_write_ienable() . . . 15-15
altera_avalon_fifo_write_almostfull() . . . 15-16
altera_avalon_fifo_write_almostempty() . . . 15-16
altera_avalon_write_fifo() . . . 15-16
altera_avalon_write_other_info() . . . 15-17
altera_avalon_fifo_read_fifo() . . . 15-17
Referenced Documents . . . 15-18
Document Revision History . . . 15-18

Chapter 16. Avalon-ST Multi-Channel Shared Memory FIFO Core

Core Overview . . . 16-1
Performance and Resource Utilization . . . 16-2
Functional Description . . . 16-3
Interfaces . . . 16-3
Avalon-ST Interfaces . . . 16-3
Avalon-MM Interfaces . . . 16-4

Operation 16-4
Instantiating the Core in SOPC Builder 16-5
Device Support 16-5
Software Programming Model 16-5
HAL System Library Support 16-5
Register Map 16-6
Control Register Interface 16-6
Fill-Level Register Interface 16-6
Referenced Documents 16-6
Document Revision History 16-7

Section III. Transport and Communication

Chapter 17. SPI Slave/JTAG to Avalon Master Bridge Cores

Core Overview 17-1
Functional Description 17-1
Instantiating the Core in SOPC Builder 17-3
Device Support 17-3
Referenced Documents 17-4
Document Revision History 17-4

Chapter 18. Avalon Streaming Channel Multiplexer and Demultiplexer Cores

Core Overview 18-1
Resource Usage and Performance 18-1
Multiplexer 18-2
Functional Description 18-2
Input Interfaces 18-3
Output Interface 18-3
Instantiating the Multiplexer in SOPC Builder 18-3
Functional Parameters 18-3
Output Interface 18-4
Demultiplexer 18-4
Functional Description 18-4
Input Interface 18-5
Output Interfaces 18-5
Instantiating the Demultiplexer in SOPC Builder 18-5
Functional Parameters 18-5
Input Interface 18-6
Device Support 18-6
Hardware Simulation Considerations 18-6
Software Programming Model 18-6
Document Revision History 18-7

Chapter 19. Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores

Core Overview 19-1
Functional Description 19-1
Interfaces 19-2
Operation—Avalon-ST Bytes to Packets Converter Core 19-2
Operation—Avalon-ST Packets to Bytes Converter Core 19-3
Instantiating the Core in SOPC Builder 19-3
Device Support 19-4
Referenced Documents 19-4
Document Revision History 19-4

Chapter 20. Avalon Packets to Transactions Converter Core

Core Overview 20-1
Functional Description 20-1
Interfaces 20-2
Operation 20-2
Packet Formats 20-2
Supported Transactions 20-3
Malformed Packets 20-3
Instantiating the Core in SOPC Builder 20-4
Device Support 20-4
Referenced Documents 20-4
Document Revision History 20-4

Chapter 21. Avalon-ST Round Robin Scheduler Core

Core Overview 21-1
Performance and Resource Utilization 21-1
Functional Description 21-2
Interfaces 21-2
Almost-Full Status Interface 21-2
Request Interface 21-3
Operations 21-3
Instantiating the Core in SOPC Builder 21-4
Device Support 21-4
Document Revision History 21-4

Chapter 22. Avalon-ST Delay Core

Core Overview 22-1
Functional Description 22-1
Reset 22-1
Interfaces 22-2
Instantiating the Core in SOPC Builder 22-2
Device Support 22-3
Referenced Documents 22-3
Document Revision History 22-3

Chapter 23. Avalon-ST Splitter Core

Core Overview 23-1
Functional Description 23-1
Backpressure 23-1
Interfaces 23-2
Instantiating the Core in SOPC Builder 23-2
Device Support 23-3
Referenced Documents 23-3
Document Revision History 23-3

Section IV. Peripherals

Chapter 24. Scatter-Gather DMA Controller Core

Core Overview 24-1
Example Systems 24-1
Comparison of SG-DMA Controller Core and DMA Controller Core 24-2
In This Chapter 24-2

Resource Usage and Performance . . . 24-3
Functional Description . . . 24-3
Functional Blocks and Configurations . . . 24-4
Descriptor Processor . . . 24-4
DMA Read Block . . . 24-4
DMA Write Block . . . 24-4
Memory-to-Memory Configuration . . . 24-5
Memory-to-Stream Configuration . . . 24-5
Stream-to-Memory Configuration . . . 24-6
DMA Descriptors . . . 24-6
Descriptor Processing . . . 24-7
Building and Updating Descriptor List . . . 24-8
Error Conditions . . . 24-8
Device Support . . . 24-9
Instantiating the Core in SOPC Builder . . . 24-9
Simulation Considerations . . . 24-10
Software Programming Model . . . 24-10
HAL System Library Support . . . 24-10
Software Files . . . 24-10
Register Maps . . . 24-11
DMA Descriptors . . . 24-13
Timeouts . . . 24-15
Programming with SG-DMA Controller . . . 24-15
Data Structure . . . 24-15
SG-DMA API . . . 24-17
alt_avalon_sgdma_do_async_transfer() . . . 24-18
alt_avalon_sgdma_do_sync_transfer() . . . 24-18
alt_avalon_sgdma_construct_mem_to_mem_desc() . . . 24-19
alt_avalon_sgdma_construct_stream_to_mem_desc() . . . 24-20
alt_avalon_sgdma_construct_mem_to_stream_desc() . . . 24-21
alt_avalon_sgdma_check_descriptor_status() . . . 24-22
alt_avalon_sgdma_register_callback() . . . 24-22
alt_avalon_sgdma_start() . . . 24-22
alt_avalon_sgdma_stop() . . . 24-23
alt_avalon_sgdma_open() . . . 24-23
Referenced Documents . . . 24-24
Document Revision History . . . 24-24

Chapter 25. DMA Controller Core

Core Overview . . . 25-1
Functional Description . . . 25-1
Setting Up DMA Transactions . . . 25-2
The Master Read and Write Ports . . . 25-3
Addressing and Address Incrementing . . . 25-3
Instantiating the Core in SOPC Builder . . . 25-4
DMA Parameters (Basic) . . . 25-4
Transfer Size . . . 25-4
Burst Transactions . . . 25-4
FIFO Depth . . . 25-4
FIFO Implementation . . . 25-5
Advanced Options . . . 25-5
Allowed Transactions . . . 25-5
Device Support . . . 25-5
Software Programming Model . . . 25-5

HAL System Library Support 25-5
ioctl() Operations 25-6
Limitations 25-6
Software Files 25-6
Register Map 25-7
status Register 25-7
readaddress Register 25-8
writeaddress Register 25-8
length Register 25-8
control Register 25-8
Interrupt Behavior 25-10
Referenced Documents 25-10
Document Revision History 25-10

Chapter 26. Video Sync Generator and Pixel Converter Cores

Core Overview 26-1
Video Sync Generator 26-2
Functional Description 26-2
Instantiating the Core in SOPC Builder 26-3
Signals 26-4
Timing Diagrams 26-4
Pixel Converter 26-5
Functional Description 26-5
Instantiating the Core in SOPC Builder 26-5
Signals 26-6
Device Support 26-6
Hardware Simulation Considerations 26-6
Referenced Documents 26-6
Document Revision History 26-7

Chapter 27. Interval Timer Core

Core Overview 27-1
Functional Description 27-1
Avalon-MM Slave Interface 27-2
Device Support 27-2
Instantiating the Core in SOPC Builder 27-3
Timeout Period 27-3
Counter Size 27-3
Hardware Options 27-3
Register Options 27-4
Output Signal Options 27-4
Configuring the Timer as a Watchdog Timer 27-4
Software Programming Model 27-5
HAL System Library Support 27-5
System Clock Driver 27-5
Timestamp Driver 27-5
Limitations 27-6
Software Files 27-6
Register Map 27-6
status Register 27-7
control Register 27-7
period_n Registers 27-8
snap_n Registers 27-8

Interrupt Behavior 27-8
Referenced Documents 27-8
Document Revision History 27-9

Chapter 28. Mutex Core
Core Overview 28-1
Functional Description 28-1
Device Support 28-2
Instantiating the Core in SOPC Builder 28-2
Software Programming Model 28-2
Software Files 28-2
Hardware Access Routines 28-3
Mutex API 28-4
altera_avalon_mutex_is_mine() 28-4
altera_avalon_mutex_first_lock() 28-4
altera_avalon_mutex_lock() 28-4
altera_avalon_mutex_open() 28-5
altera_avalon_mutex_trylock() 28-5
altera_avalon_mutex_unlock() 28-5
Document Revision History 28-6

Chapter 29. Mailbox Core
Core Overview 29-1
Functional Description 29-1
Device Support 29-2
Instantiating the Core in SOPC Builder 29-2
Software Programming Model 29-3
Software Files 29-3
Programming with the Mailbox Core 29-3
Mailbox API 29-5
altera_avalon_mailbox_close() 29-5
altera_avalon_mailbox_get() 29-5
altera_avalon_mailbox_open() 29-5
altera_avalon_mailbox_pend() 29-6
altera_avalon_mailbox_post() 29-6
Document Revision History 29-6

Chapter 30. Vectored Interrupt Controller Core
Core Overview 30-1
Functional Description 30-2
External Interfaces 30-2
clk 30-2
irq_input 30-2
interrupt_controller_out 30-3
interrupt_controller_in 30-3
csr_access 30-3
Functional Blocks 30-4
Interrupt Request Block 30-4
Priority Processing Block 30-4
Vector Generation Block 30-5
Daisy Chaining VIC Cores 30-5
Latency Information 30-6
Register Maps 30-6

Device Support 30-9
Instantiating the Core in SOPC Builder 30-9
Altera HAL Software Programming Model 30-10
Software Files 30-10
Macros 30-11
Data Structure 30-11
VIC API 30-12
alt_vic_sw_interrupt_set() 30-12
alt_vic_sw_interrupt_clear() 30-13
alt_vic_sw_interrupt_status() 30-13
alt_vic_irq_set_level() 30-14
Run-time Initialization 30-14
Board Support Package 30-14
VIC BSP Settings 30-14
Default Settings for RRS and RIL 30-18
VIC BSP Design Rules for Altera Hal Implementation 30-18
RTOS Considerations 30-19
Referenced Documents 30-19
Document Revision History 30-20

Section V. Test and Debug Peripherals

Chapter 31. Avalon-ST JTAG Interface Core

Core Overview 31-1
Functional Description 31-1
Interfaces 31-2
Special characters 31-2
Operation 31-2
Instantiating the Core in SOPC Builder 31-3
Device Support 31-3
Referenced Documents 31-3
Document Revision History 31-3

Chapter 32. System ID Core

Core Overview 32-1
Functional Description 32-1
Device Support 32-2
Instantiating the Core in SOPC Builder 32-2
Software Programming Model 32-2
alt_avalon_sysid_test() 32-2
Document Revision History 32-3

Chapter 33. Performance Counter Core

Core Overview 33-1
Functional Description 33-2
Section Counters 33-2
Global Counter 33-2
Register Map 33-2
System Reset 33-3
Device and Tools Support 33-3
Instantiating the Core in SOPC Builder 33-3
Define Counters 33-3
Multiple Clock Domain Considerations 33-3

Hardware Simulation Considerations . . . 33-4
Software Programming Model . . . 33-4
Software Files . . . 33-4
Using the Performance Counter . . . 33-4
API Summary . . . 33-4
Startup . . . 33-5
Global Counter Usage . . . 33-5
Section Counter Usage . . . 33-5
Viewing Counter Values . . . 33-5
Interrupt Behavior . . . 33-6
Performance Counter API . . . 33-7
PERF_RESET() . . . 33-7
PERF_START_MEASURING() . . . 33-7
PERF_STOP_MEASURING() . . . 33-7
PERF_BEGIN() . . . 33-8
PERF_END() . . . 33-8
perf_print_formatted_report() . . . 33-9
perf_get_total_time() . . . 33-9
perf_get_section_time() . . . 33-10
perf_get_num_starts() . . . 33-10
alt_get_cpu_freq() . . . 33-10
Referenced Documents . . . 33-11
Document Revision History . . . 33-11

Chapter 34. Avalon Streaming Test Pattern Generator and Checker Cores

Core Overview . . . 34-1
Resource Utilization and Performance . . . 34-1
Test Pattern Generator . . . 34-3
Functional Description . . . 34-3
Command Interface . . . 34-3
Control and Status Interface . . . 34-4
Output Interface . . . 34-4
Instantiating the Test Pattern Generator in SOPC Builder . . . 34-4
Functional Parameter . . . 34-4
Output Interface . . . 34-4
Test Pattern Checker . . . 34-5
Functional Description . . . 34-5
Input Interface . . . 34-5
Control and Status Interface . . . 34-6
Instantiating the Test Pattern Checker in SOPC Builder . . . 34-6
Functional Parameter . . . 34-6
Input Parameters . . . 34-6
Device Support . . . 34-6
Hardware Simulation Considerations . . . 34-6
Software Programming Model . . . 34-7
HAL System Library Support . . . 34-7
Software Files . . . 34-7
Register Maps . . . 34-8
Test Pattern Generator Control and Status Registers . . . 34-8
Test Pattern Generator Command Registers . . . 34-9
Test Pattern Checker Control and Status Registers . . . 34-10
Test Pattern Generator API . . . 34-12
data_source_reset() . . . 34-12
data_source_init() . . . 34-12

data_source_get_id() 34-12
data_source_get_supports_packets() 34-13
data_source_get_num_channels() 34-13
data_source_get_symbols_per_cycle() 34-13
data_source_set_enable() 34-13
data_source_get_enable() 34-14
data_source_set_throttle() 34-14
data_source_get_throttle() 34-14
data_source_is_busy() 34-14
data_source_fill_level() 34-15
data_source_send_data() 34-15
Test Pattern Checker API 34-16
data_sink_reset() 34-16
data_sink_init() 34-16
data_sink_get_id() 34-16
data_sink_get_supports_packets() 34-16
data_sink_get_num_channels() 34-17
data_sink_get_symbols_per_cycle() 34-17
data_sink_set enable() 34-17
data_sink_get_enable() 34-17
data_sink_set_throttle() 34-18
data_sink_get_throttle() 34-18
data_sink_get_packet_count() 34-18
data_sink_get_symbol_count() 34-18
data_sink_get_error_count() 34-19
data_sink_get_exception() 34-19
data_sink_exception_is_exception() 34-19
data_sink_exception_has_data_error() 34-19
data_sink_exception_has_missing_sop() 34-20
data_sink_exception_has_missing_eop() 34-20
data_sink_exception_signalled_error() 34-20
data_sink_exception_channel() 34-20
Document Revision History 34-21

Chapter 35. Avalon Streaming Data Pattern Generator and Checker Cores

Core Overview 35-1
Data Pattern Generator 35-1
Functional Description 35-1
Control and Status Interface 35-2
Output Interface 35-2
Supported Data Patterns 35-2
Inject Error 35-3
Preamble Mode 35-3
Instantiating the Data Pattern Generator in SOPC Builder 35-3
Output Parameter 35-3
Data Pattern Checker 35-4
Functional Description 35-4
Control and Status Interface 35-4
Input Interface 35-5
Supported Data Patterns 35-5
Lock 35-5
Bit and Error Counters 35-5
Instantiating the Data Pattern Checker in SOPC Builder 35-5
Input Parameter 35-5

Device Support 35-6
Hardware Simulation Considerations 35-6
Software Programming Model 35-6
Register Maps 35-6
Data Pattern Generator Control Registers 35-6
Data Pattern Checker Control and Status Registers 35-8
Referenced Documents 35-10
Document Revision History 35-10

Section VI. Clock Control Peripherals

Chapter 36. PLL Cores

Core Overview 36-1
Functional Description 36-1
ALTPLL Megafunction 36-2
Clock Outputs 36-2
PLL Status and Control Signals 36-3
System Reset Considerations 36-3
Device Support 36-3
Instantiating the Cores in SOPC Builder 36-3
Instantiating the Avalon ALTPLL Core 36-3
Instantiating the PLL Core 36-3
PLL Settings Page 36-4
Interface Page 36-4
Finish 36-4
Hardware Simulation Considerations 36-5
Register Definitions and Bit List 36-5
Status Register 36-5
Control Register 36-6
Phase Reconfig Control Register 36-6
Referenced Documents 36-7
Document Revision History 36-7

Additional Information

About this Handbook Info-1
How to Contact Altera Info-1
Third-Party Software Product Information Info-1
Typographic Conventions Info-2

Section I. Off-Chip Interface Peripherals

This section describes the interfaces to off-chip devices provided for SOPC Builder systems.

This section includes the following chapters:

- Chapter 1, SDRAM Controller Core
- Chapter 2, CompactFlash Core
- Chapter 3, Common Flash Interface Controller Core
- Chapter 4, EPCS Serial Flash Controller Core
- Chapter 5, JTAG UART Core
- Chapter 6, UART Core
- Chapter 7, SPI Core
- Chapter 8, Optrex 16207 LCD Controller Core
- Chapter 9, PIO Core
- Chapter 10, Avalon-ST Serial Peripheral Interface Core
- Chapter 11, PCI Lite Core
- Chapter 12, Cyclone III Remote Update Controller Core
- Chapter 13, MDIO Core

☞ For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. SDRAM Controller Core

Core Overview

The SDRAM controller core with Avalon® interface provides an Avalon Memory-Mapped (Avalon-MM) interface to off-chip SDRAM. The SDRAM controller allows designers to create custom systems in an Altera® device that connect easily to SDRAM chips. The SDRAM controller supports standard SDRAM as described in the PC100 specification.

SDRAM is commonly used in cost-sensitive applications requiring large amounts of volatile memory. While SDRAM is relatively inexpensive, control logic is required to perform refresh operations, open-row management, and other delays and command sequences. The SDRAM controller connects to one or more SDRAM chips, and handles all SDRAM protocol requirements. Internal to the device, the core presents an Avalon-MM slave port that appears as linear memory (flat address space) to Avalon-MM master peripherals.

The core can access SDRAM subsystems with various data widths (8, 16, 32, or 64 bits), various memory sizes, and multiple chip selects. The Avalon-MM interface is latency-aware, allowing read transfers to be pipelined. The core can optionally share its address and data buses with other off-chip Avalon-MM tri-state devices. This feature is valuable in systems that have limited I/O pins, yet must connect to multiple memory chips in addition to SDRAM.

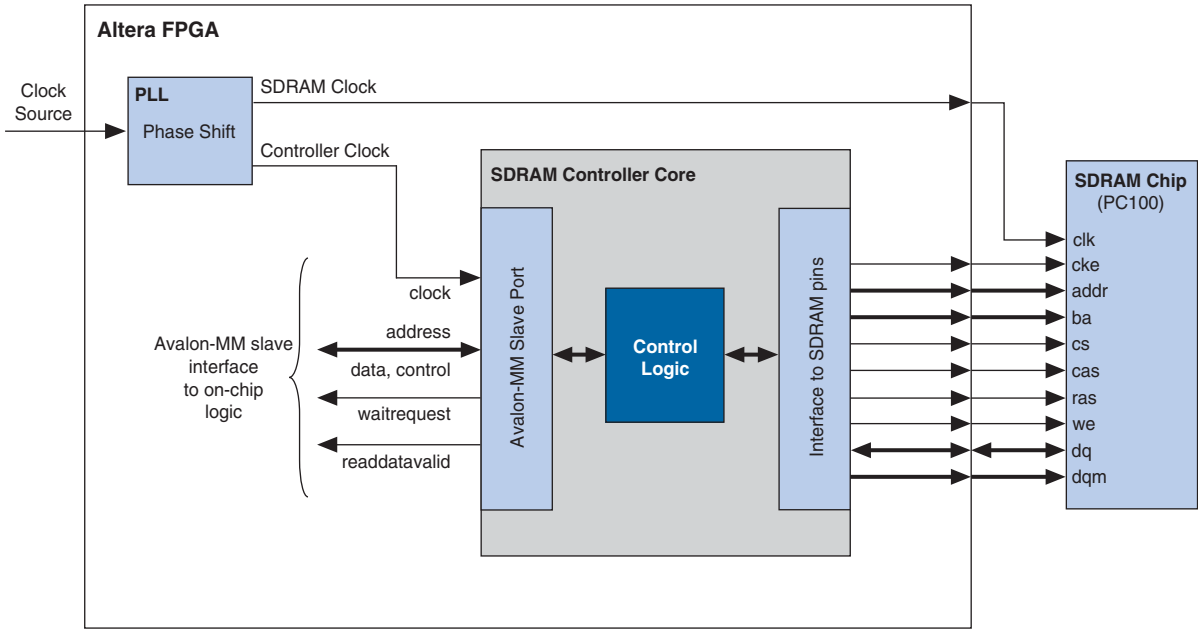
The SDRAM controller core with Avalon interface is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 1–2
- "Device Support" on page 1–5
- "Instantiating the Core in SOPC Builder" on page 1–5
- "Hardware Simulation Considerations" on page 1–7
- "Software Programming Model" on page 1–10
- "Clock, PLL and Timing Considerations" on page 1–10

Functional Description

Figure 1–1 shows a block diagram of the SDRAM controller core connected to an external SDRAM chip.

Figure 1–1. SDRAM Controller with Avalon Interface Block Diagram



The following sections describe the components of the SDRAM controller core in detail. All options are specified at system generation time, and cannot be changed at runtime.

Avalon-MM Interface

The Avalon-MM slave port is the user-visible part of the SDRAM controller core. The slave port presents a flat, contiguous memory space as large as the SDRAM chip(s). When accessing the slave port, the details of the PC100 SDRAM protocol are entirely transparent. The Avalon-MM interface behaves as a simple memory interface. There are no memory-mapped configuration registers.

The Avalon-MM slave port supports peripheral-controlled wait states for read and write transfers. The slave port stalls the transfer until it can present valid data. The slave port also supports read transfers with variable latency, enabling high-bandwidth, pipelined read transfers. When a master peripheral reads sequential addresses from the slave port, the first data returns after an initial period of latency. Subsequent reads can produce new data every clock cycle. However, data is not guaranteed to return every clock cycle, because the SDRAM controller must pause periodically to refresh the SDRAM.

For details about Avalon-MM transfer types, refer to the *Avalon Interface Specifications*.

Off-Chip SDRAM Interface

The interface to the external SDRAM chip presents the signals defined by the PC100 standard. These signals must be connected externally to the SDRAM chip(s) through I/O pins on the Altera device.

Signal Timing and Electrical Characteristics

The timing and sequencing of signals depends on the configuration of the core. The hardware designer configures the core to match the SDRAM chip chosen for the system. See "Instantiating the Core in SOPC Builder" on page 1–5 for details. The electrical characteristics of the device pins depend on both the target device family and the assignments made in the Quartus® II software. Some device families support a wider range of electrical standards, and therefore are capable of interfacing with a greater variety of SDRAM chips. For details, refer to the device handbook for the target device family.

Synchronizing Clock and Data Signals

The clock for the SDRAM chip (SDRAM clock) must be driven at the same frequency as the clock for the Avalon-MM interface on the SDRAM controller (controller clock). As in all synchronous designs, you must ensure that address, data, and control signals at the SDRAM pins are stable when a clock edge arrives. As shown in Figure 1–1, you can use an on-chip phase-locked loop (PLL) to alleviate clock skew between the SDRAM controller core and the SDRAM chip. At lower clock speeds, the PLL might not be necessary. At higher clock rates, a PLL is necessary to ensure that the SDRAM clock toggles only when signals are stable on the pins. The PLL block is not part of the SDRAM controller core. If a PLL is necessary, you must instantiate it manually. You can instantiate the PLL core interface, which is an SOPC Builder component, or instantiate an ALTPLL megafunction outside the SOPC Builder system module.

If you use a PLL, you must tune the PLL to introduce a clock phase shift so that SDRAM clock edges arrive after synchronous signals have stabilized. See "Clock, PLL and Timing Considerations" on page 1–10 for details.

For more information about instantiating a PLL in your SOPC Builder system, refer to "PLL Cores" on page 36–1. The Nios® II development tools provide example hardware designs that use the SDRAM controller core in conjunction with a PLL, which you can use as a reference for your custom designs. The Nios II development tools are available free for download from www.altera.com.

Clock Enable (CKE) Not Supported

The SDRAM controller does not support clock-disable modes. The SDRAM controller permanently asserts the CKE signal on the SDRAM.

Sharing Pins with Other Avalon-MM Tri-State Devices

If an Avalon-MM tri-state bridge is present in the SOPC Builder system, the SDRAM controller core can share pins with the existing tri-state bridge. In this case, the core's `addr`, `dq` (data) and `dqm` (byte-enable) pins are shared with other devices connected to the Avalon-MM tri-state bridge. This feature conserves I/O pins, which is valuable in systems that have multiple external memory chips (for example, flash, SRAM, and SDRAM), but too few pins to dedicate to the SDRAM chip. See "Performance Considerations" for details about how pin sharing affects performance.

☞ The SDRAM addresses must connect all address bits regardless of the size of the word so that the low-order address bits on the tri-state bridge align with the low-order address bits on the memory device. The Avalon-MM tristate address signal always presents a byte address. It is not possible to drop A0 of the tri-state bridge for memories when the smallest access size is 16 bits or A0-A1 of the tri-state bridge when the smallest access size is 32 bits.

Board Layout and Pinout Considerations

When making decisions about the board layout and device pinout, try to minimize the skew between the SDRAM signals. For example, when assigning the device pinout, group the SDRAM signals, including the SDRAM clock output, physically close together. Also, you can use the **Fast Input Register** and **Fast Output Register** logic options in the Quartus II software. These logic options place registers for the SDRAM signals in the I/O cells. Signals driven from registers in I/O cells have similar timing characteristics, such as t_{CO}, t_{SU}, and t_H.

Performance Considerations

Under optimal conditions, the SDRAM controller core's bandwidth approaches one word per clock cycle. However, because of the overhead associated with refreshing the SDRAM, it is impossible to reach one word per clock cycle. Other factors affect the core's performance, as described in the following sections.

Open Row Management

SDRAM chips are arranged as multiple banks of memory, in which each bank is capable of independent open-row address management. The SDRAM controller core takes advantage of open-row management for a single bank. Continuous reads or writes within the same row and bank operate at rates approaching one word per clock. Applications that frequently access different destination banks require extra management cycles to open and close rows.

Sharing Data and Address Pins

When the controller shares pins with other tri-state devices, average access time usually increases and bandwidth decreases. When access to the tri-state bridge is granted to other devices, the SDRAM incurs overhead to open and close rows. Furthermore, the SDRAM controller has to wait several clock cycles before it is granted access again.

To maximize bandwidth, the SDRAM controller automatically maintains control of the tri-state bridge as long as back-to-back read or write transactions continue within the same row and bank.

☞ This behavior may degrade the average access time for other devices sharing the Avalon-MM tri-state bridge.

The SDRAM controller closes an open row whenever there is a break in back-to-back transactions, or whenever a refresh transaction is required. As a result:

- The controller cannot permanently block access to other devices sharing the tri-state bridge.

- The controller is guaranteed not to violate the SDRAM's row open time limit.

Hardware Design and Target Device

The target device affects the maximum achievable clock frequency of a hardware design. Certain device families achieve higher f_{MAX} performance than other families. Furthermore, within a device family, faster speed grades achieve higher performance. The SDRAM controller core can achieve 100 MHz in Altera's high-performance device families, such as Stratix® series. However, the core might not achieve 100 MHz performance in all Altera device families.

The f_{MAX} performance also depends on the SOPC Builder system design. The SDRAM controller clock can also drive other logic in the system module, which might affect the maximum achievable frequency. For the SDRAM controller core to achieve f_{MAX} performance of 100 MHz, all components driven by the same clock must be designed for a 100 MHz clock rate, and timing analysis in the Quartus II software must verify that the overall hardware design is capable of 100 MHz operation.

Device Support

The SDRAM Controller with Avalon interface core supports all Altera device families. Different device families support different I/O standards, which may affect the ability of the core to interface to certain SDRAM chips. For details about supported I/O types, refer to the device handbook for the target device family.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the SDRAM controller in SOPC Builder to specify hardware and simulation features. The SDRAM controller MegaWizard has two pages: **Memory Profile** and **Timing**. This section describes the options available on each page.

The **Presets** list offers several pre-defined SDRAM configurations as a convenience. If the SDRAM subsystem on the target board matches one of the preset configurations, you can configure the SDRAM controller core easily by selecting the appropriate preset value. The following preset configurations are defined:

- Micron MT8LSDT1664HG module
- Four SDR100 8 MByte × 16 chips
- Single Micron MT48LC2M32B2-7 chip
- Single Micron MT48LC4M32B2-7 chip
- Single NEC D4564163-A80 chip (64 MByte × 16)
- Single Alliance AS4LC1M16S1-10 chip
- Single Alliance AS4LC2M8S0-10 chip

Selecting a preset configuration automatically changes values on the **Memory Profile** and **Timing** tabs to match the specific configuration. Altering a configuration setting on any page changes the **Preset** value to **custom**.

Memory Profile Page

The **Memory Profile** page allows you to specify the structure of the SDRAM subsystem such as address and data bus widths, the number of chip select signals, and the number of banks. Table 1–1 lists the settings available on the **Memory Profile** page.

Table 1–1. Memory Profile Page Settings

Settings		Allowed Values	Default Values	Description
Data Width		8, 16, 32, 64	32	SDRAM data bus width. This value determines the width of the `dq` bus (data) and the `dqm` bus (byte-enable).
Architecture Settings	Chip Selects	1, 2, 4, 8	1	Number of independent chip selects in the SDRAM subsystem. By using multiple chip selects, the SDRAM controller can combine multiple SDRAM chips into one memory subsystem.
	Banks	2, 4	4	Number of SDRAM banks. This value determines the width of the `ba` bus (bank address) that connects to the SDRAM. The correct value is provided in the data sheet for the target SDRAM.
Address Width Settings	Row	11, 12, 13, 14	12	Number of row address bits. This value determines the width of the `addr` bus. The Row and Column values depend on the geometry of the chosen SDRAM. For example, an SDRAM organized as 4096 (2^{12}) rows by 512 columns has a Row value of 12.
	Column	>= 8, and less than Row value	8	Number of column address bits. For example, the SDRAM organized as 4096 rows by 512 (2^9) columns has a Column value of 9.
Share pins via tri-state bridge dq/dqm/addr I/O pins		On, Off	Off	When set to No, all pins are dedicated to the SDRAM chip. When set to Yes, the `addr`, `dq`, and `dqm` pins can be shared with a tristate bridge in the system. In this case, select the appropriate tristate bridge from the pull-down menu.
Include a functional memory model in the system testbench		On, Off	On	When on, SOPC Builder creates a functional simulation model for the SDRAM chip. This default memory model accelerates the process of creating and verifying systems that use the SDRAM controller. See "Hardware Simulation Considerations" on page 1–7.

Based on the settings entered on the **Memory Profile** page, the wizard displays the expected memory capacity of the SDRAM subsystem in units of megabytes, megabits, and number of addressable words. Compare these expected values to the actual size of the chosen SDRAM to verify that the settings are correct.

Timing Page

The **Timing** page allows designers to enter the timing specifications of the SDRAM chip(s) used. The correct values are available in the manufacturer's data sheet for the target SDRAM. Table 1–2 lists the settings available on the **Timing** page.

Table 1–2. Timing Page Settings

Settings	Allowed Values	Default Value	Description
CAS latency	1, 2, 3	3	Latency (in clock cycles) from a read command to data out.
Initialization refresh cycles	1–8	2	This value specifies how many refresh cycles the SDRAM controller performs as part of the initialization sequence after reset.
Issue one refresh command every	—	15.625 µs	This value specifies how often the SDRAM controller refreshes the SDRAM. A typical SDRAM requires 4,096 refresh commands every 64 ms, which can be achieved by issuing one refresh command every 64 ms / 4,096 = 15.625 µs.
Delay after power up, before initialization	—	100 µs	The delay from stable clock and power to SDRAM initialization.
Duration of refresh command (t_rfc)	—	70 ns	Auto Refresh period.
Duration of precharge command (t_rp)	—	20 ns	Precharge command period.
ACTIVE to READ or WRITE delay (t_rcd)	—	20 ns	ACTIVE to READ or WRITE delay.
Access time (t_ac)	—	17 ns	Access time from clock edge. This value may depend on CAS latency.
Write recovery time (t_wr, No auto precharge)	—	14 ns	Write recovery if explicit precharge commands are issued. This SDRAM controller always issues explicit precharge commands.

Regardless of the exact timing values you specify, the actual timing achieved for each parameter is an integer multiple of the Avalon clock period. For the **Issue one refresh command every** parameter, the actual timing is the greatest number of clock cycles that does not exceed the target value. For all other parameters, the actual timing is the smallest number of clock ticks that provides a value greater than or equal to the target value.

Hardware Simulation Considerations

This section discusses considerations for simulating systems with SDRAM. Three major components are required for simulation:

- A simulation model for the SDRAM controller.
- A simulation model for the SDRAM chip(s), also called the memory model.
- A simulation testbench that wires the memory model to the SDRAM controller pins.

Some or all of these components are generated by SOPC Builder at system generation time.

SDRAM Controller Simulation Model

The SDRAM controller design files generated by SOPC Builder are suitable for both synthesis and simulation. Some simulation features are implemented in the HDL using "translate on/off" synthesis directives that make certain sections of HDL code invisible to the synthesis tool.

The simulation features are implemented primarily for easy simulation of Nios and Nios II processor systems using the ModelSim® simulator. The SDRAM controller simulation model is not ModelSim specific. However, minor changes may be required to make the model work with other simulators.

CAUTION

If you change the simulation directives to create a custom simulation flow, be aware that SOPC Builder overwrites existing files during system generation. Take precautions to ensure your changes are not overwritten.

Refer to *AN 351: Simulating Nios II Processor Designs* for a demonstration of simulation of the SDRAM controller in the context of Nios II embedded processor systems.

SDRAM Memory Model

This section describes the two options for simulating a memory model of the SDRAM chip(s).

Using the Generic Memory Model

If the **Include a functional memory model the system testbench** option is enabled at system generation, SOPC Builder generates an HDL simulation model for the SDRAM memory. In the auto-generated system testbench, SOPC Builder automatically wires this memory model to the SDRAM controller pins.

Using the automatic memory model and testbench accelerates the process of creating and verifying systems that use the SDRAM controller. However, the memory model is a generic functional model that does not reflect the true timing or functionality of real SDRAM chips. The generic model is always structured as a single, monolithic block of memory. For example, even for a system that combines two SDRAM chips, the generic memory model is implemented as a single entity.

Using the SDRAM Manufacturer's Memory Model

If the **Include a functional memory model the system testbench** option is not enabled, you are responsible for obtaining a memory model from the SDRAM manufacturer, and manually wiring the model to the SDRAM controller pins in the system testbench.

Example Configurations

The following examples show how to connect the SDRAM controller outputs to an SDRAM chip or chips. The bus labeled `ctl` is an aggregate of the remaining signals, such as `cas_n`, `ras_n`, `cke` and `we_n`.

Figure 1–2 shows a single 128-Mbit SDRAM chip with 32-bit data. The address, data, and control signals are wired directly from the controller to the chip. The result is a 128-Mbit (16-Mbyte) memory space.

Figure 1–2. Single 128-Mbit SDRAM Chip with 32-Bit Data

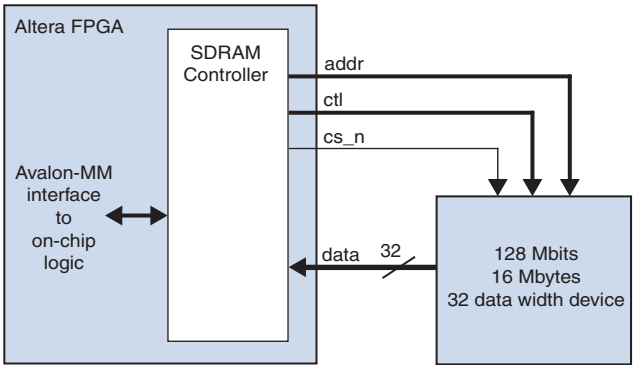


Figure 1–3 shows two 64-Mbit SDRAM chips, each with 16-bit data. The address and control signals connect in parallel to both chips. The chips share the chipselect (`cs_n`) signal. Each chip provides half of the 32-bit data bus. The result is a logical 128-Mbit (16-Mbyte) 32-bit data memory.

Figure 1–3. Two 64-MBit SDRAM Chips Each with 16-Bit Data

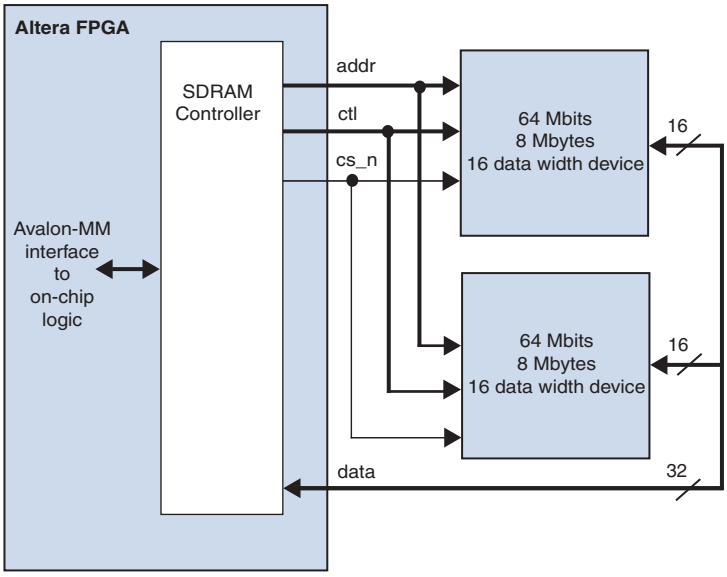
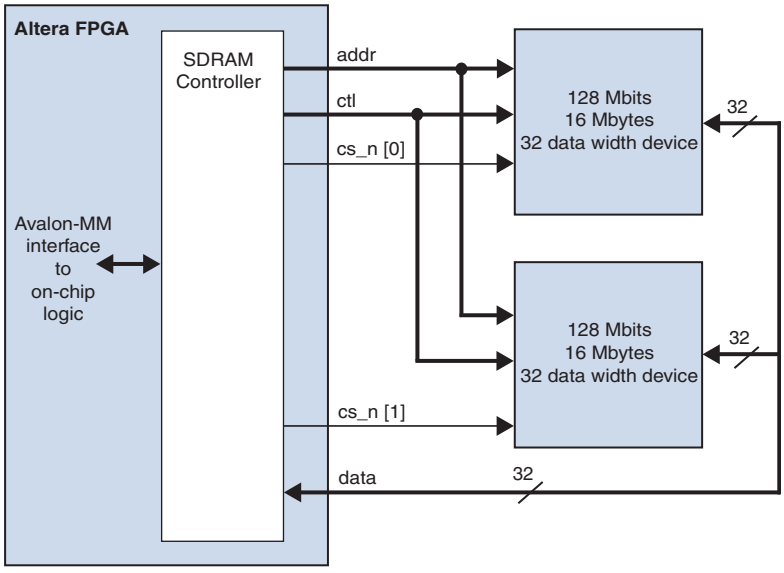


Figure 1–4 shows two 128-Mbit SDRAM chips, each with 32-bit data. The address, data, and control signals connect in parallel to the two chips. The chipselect bus (`cs_n[1:0]`) determines which chip is selected. The result is a logical 256-Mbit 32-bit wide memory.

Figure 1–4. Two 128-Mbit SDRAM Chips Each with 32-Bit Data



Software Programming Model

The SDRAM controller behaves like simple memory when accessed via the Avalon-MM interface. There are no software-configurable settings and no memory-mapped registers. No software driver routines are required for a processor to access the SDRAM controller.

Clock, PLL and Timing Considerations

This section describes issues related to synchronizing signals from the SDRAM controller core with the clock that drives the SDRAM chip. During SDRAM transactions, the address, data, and control signals are valid at the SDRAM pins for a window of time, during which the SDRAM clock must toggle to capture the correct values. At slower clock frequencies, the clock naturally falls within the valid window. At higher frequencies, you must compensate the SDRAM clock to align with the valid window.

Determine when the valid window occurs either by calculation or by analyzing the SDRAM pins with an oscilloscope. Then use a PLL to adjust the phase of the SDRAM clock so that edges occur in the middle of the valid window. Tuning the PLL might require trial-and-error effort to align the phase shift to the properties of your target board.

For details about the PLL circuitry in your target device, refer to the appropriate device family handbook. For details about configuring the PLLs in Altera devices, refer to the *ALTPLL Megafunction User Guide*.

Factors Affecting SDRAM Timing

The location and duration of the window depends on several factors:

- Timing parameters of the device and SDRAM I/O pins — I/O timing parameters vary based on device family and speed grade.
- Pin location on the device — I/O pins connected to row routing have different timing than pins connected to column routing.
- Logic options used during the Quartus II compilation — Logic options such as the **Fast Input Register** and **Fast Output Register** logic affect the design fit. The location of logic and registers inside the device affects the propagation delays of signals to the I/O pins.
- SDRAM CAS latency

As a result, the valid window timing is different for different combinations of FPGA and SDRAM devices. The window depends on the Quartus II software fitting results and pin assignments.

Symptoms of an Untuned PLL

Detecting when the PLL is not tuned correctly might be difficult. Data transfers to or from the SDRAM might not fail universally. For example, individual transfers to the SDRAM controller might succeed, whereas burst transfers fail. For processor-based systems, if software can perform read or write data to SDRAM, but cannot run when the code is located in SDRAM, the PLL is probably tuned incorrectly.

Estimating the Valid Signal Window

This section describes how to estimate the location and duration of the valid signal window using timing parameters provided in the SDRAM datasheet and the Quartus II software compilation report. After finding the window, tune the PLL so that SDRAM clock edges occur exactly in the middle of the window.

Calculating the window is a two-step process. First, determine by how much time the SDRAM clock can lag the controller clock, and then by how much time it can lead. After finding the maximum lag and lead values, calculate the midpoint between them.

☞ These calculations provide an estimation only. The following delays can also affect proper PLL tuning, but are not accounted for by these calculations.

- Signal skew due to delays on the printed circuit board — These calculations assume zero skew.
- Delay from the PLL clock output nodes to destinations — These calculations assume that the delay from the PLL SDRAM-clock output-node to the pin is the same as the delay from the PLL controller-clock output-node to the clock inputs in the SDRAM controller. If these clock delays are significantly different, you must account for this phase shift in your window calculations.

Figure 1–5 shows how to calculate the maximum length of time that the SDRAM clock can lag the controller clock, and Figure 1–6 shows how to calculate the maximum lead. Lag is a negative time shift, relative to the controller clock, and lead is a positive time shift. The SDRAM clock can lag the controller clock by the lesser of the maximum lag for a read cycle or that for a write cycle. In other words, *Maximum Lag* = minimum(*Read Lag*, *Write Lag*). Similarly, the SDRAM clock can lead by the lesser of the maximum lead for a read cycle or for a write cycle. In other words, *Maximum Lead* = minimum(*Read Lead*, *Write Lead*).

Figure 1–5. Calculating the Maximum SDRAM Clock Lag

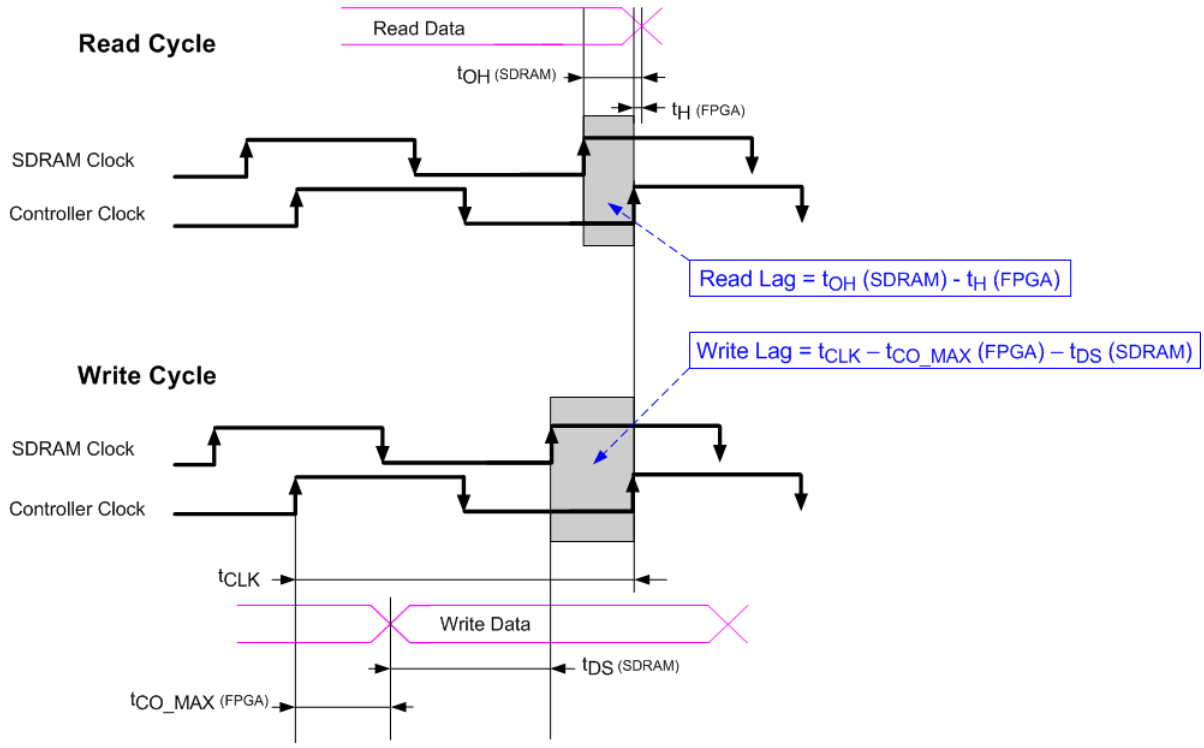
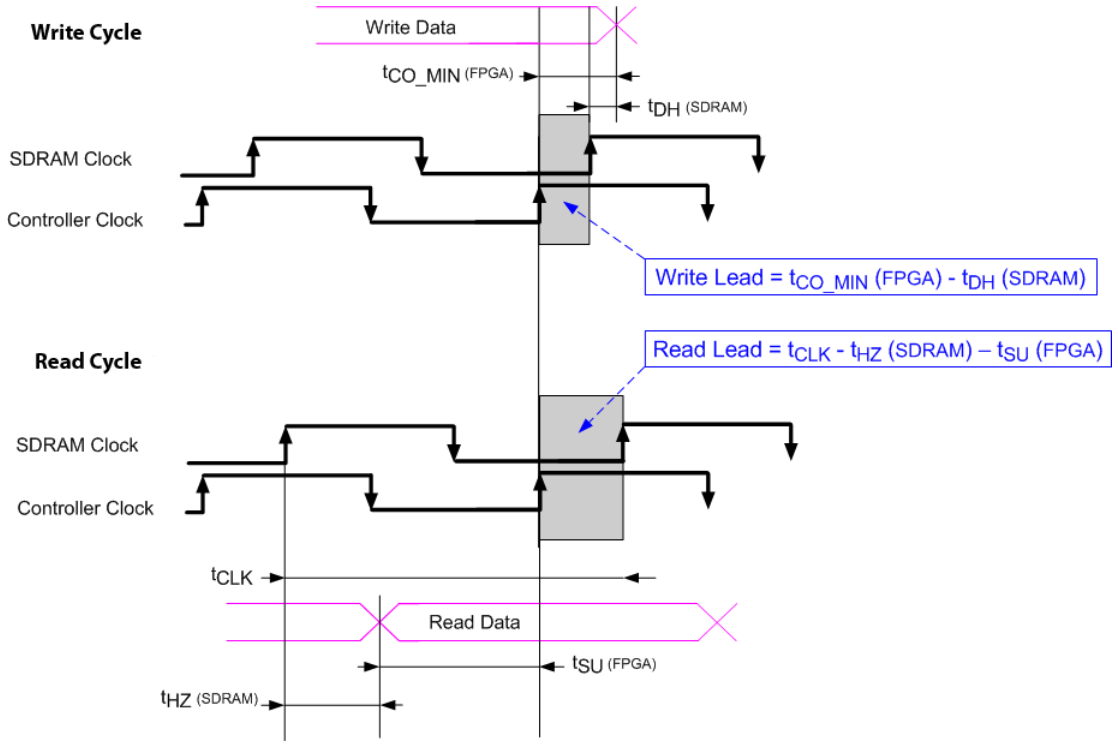


Figure 1–6. Calculating the Maximum SDRAM Clock Lead



Example Calculation

This section demonstrates a calculation of the signal window for a Micron MT48LC4M32B2-7 SDRAM chip and design targeting the Stratix II EP2S60F672C5 device. This example uses a CAS latency (CL) of 3 cycles, and a clock frequency of 50 MHz. All SDRAM signals on the device are registered in I/O cells, enabled with the **Fast Input Register** and **Fast Output Register** logic options in the Quartus II software.

Table 1–3 shows the relevant timing parameters excerpted from the MT48LC4M32B2 device datasheet.

Table 1–3. Timing Parameters for Micron MT48LC4M32B2 SDRAM Device (Part 1 of 2)

Parameter		Symbol	Value (ns) in -7 Speed Grade	
			Min.	Max.
Access time from CLK (pos. edge)	CL = 3	$t_{AC(3)}$	—	5.5
	CL = 2	$t_{AC(2)}$	—	8
	CL = 1	$t_{AC(1)}$	—	17
Address hold time		t_{AH}	1	—
Address setup time		t_{AS}	2	—
CLK high-level width		t_{CH}	2.75	—
CLK low-level width		t_{CL}	2.75	—

Table 1–3. Timing Parameters for Micron MT48LC4M32B2 SDRAM Device (Part 2 of 2)

Parameter		Symbol	Value (ns) in -7 Speed Grade	
			Min.	Max.
Clock cycle time	CL = 3	$t_{CK(3)}$	7	—
	CL = 2	$t_{CK(2)}$	10	—
	CL = 1	$t_{CK(1)}$	20	—
CKE hold time		t_{CKH}	1	—
CKE setup time		t_{CKS}	2	—
CS#, RAS#, CAS#, WE#, DQM hold time		t_{CMH}	1	—
CS#, RAS#, CAS#, WE#, DQM setup time		t_{CMS}	2	—
Data-in hold time		t_{DH}	1	
Data-in setup time		t_{DS}	2	
Data-out high-impedance time	CL = 3	$t_{HZ(3)}$		5.5
	CL = 2	$t_{HZ(2)}$	—	8
	CL = 1	$t_{HZ(1)}$	—	17
Data-out low-impedance time		t_{LZ}	1	—
Data-out hold time		t_{OH}	2.5	

Table 1–4 shows the relevant timing information, obtained from the Timing Analyzer section of the Quartus II Compilation Report. The values in the table are the maximum or minimum values among all device pins related to the SDRAM. The variance in timing between the SDRAM pins on the device is small (less than 100 ps) because the registers for these signals are placed in the I/O cell.

Table 1–4. FPGA I/O Timing Parameters

Parameter	Symbol	Value (ns)
Clock period	t_{CLK}	20
Minimum clock-to-output time	t_{CO_MIN}	2.399
Maximum clock-to-output time	t_{CO_MAX}	2.477
Maximum hold time after clock	t_{H_MAX}	–5.607
Maximum setup time before clock	t_{SU_MAX}	5.936

You must compile the design in the Quartus II software to obtain the I/O timing information for the design. Although Altera device family datasheets contain generic I/O timing information for each device, the Quartus II Compilation Report provides the most precise timing information for your specific design.

CAUTION

The timing values found in the compilation report can change, depending on fitting, pin location, and other Quartus II logic settings. When you recompile the design in the Quartus II software, verify that the I/O timing has not changed significantly.

The following examples illustrate the calculations from Figure 1–5 and Figure 1–6 using the values from Table 1–3 and Table 1–4.

The SDRAM clock can lag the controller clock by the lesser of *Read Lag* or *Write Lag*:

Read Lag = t_{OH}(SDRAM) – t_{H_MAX}(FPGA)

= 2.5 ns – (–5.607 ns) = 8.107 ns

or

Write Lag= t_{CLK} – t_{CO_MAX}(FPGA) – t_{DS}(SDRAM)

= 20 ns – 2.477 ns – 2 ns = 15.523 ns

The SDRAM clock can lead the controller clock by the lesser of *Read Lead* or *Write Lead*:

Read Lead= t_{CO_MIN}(FPGA) – t_{DH}(SDRAM)

= 2.399 ns – 1.0 ns = 1.399 ns

or

Write Lead= t_{CLK} – $t_{HZ(3)}$(SDRAM) – t_{SU_MAX}(FPGA)

= 20 ns – 5.5 ns – 5.936 ns = 8.564 ns

Therefore, for this example you can shift the phase of the SDRAM clock from –8.107 ns to 1.399 ns relative to the controller clock. Choosing a phase shift in the middle of this window results in the value (–8.107 + 1.399)/2 = –3.35 ns.

Referenced Documents

This chapter references the following documents:

- *ALTPLL Megafunction User Guide*
- *AN 351: Simulating Nios II Processor Designs*
- *Avalon Interface Specifications*
- *"PLL Cores" on page 36–1*

Document Revision History

Table 1–5 shows the revision history for this chapter.

Table 1–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

2. CompactFlash Core

Core Overview

The CompactFlash core allows you to connect SOPC Builder systems to CompactFlash storage cards in true IDE mode by providing an Avalon® Memory-Mapped (Avalon-MM) interface to the registers on the storage cards. The core supports PIO mode 0.

The CompactFlash core also provides an Avalon-MM slave interface which can be used by Avalon-MM master peripherals such as a Nios® II processor to communicate with the CompactFlash core and manage its operations.

The CompactFlash core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

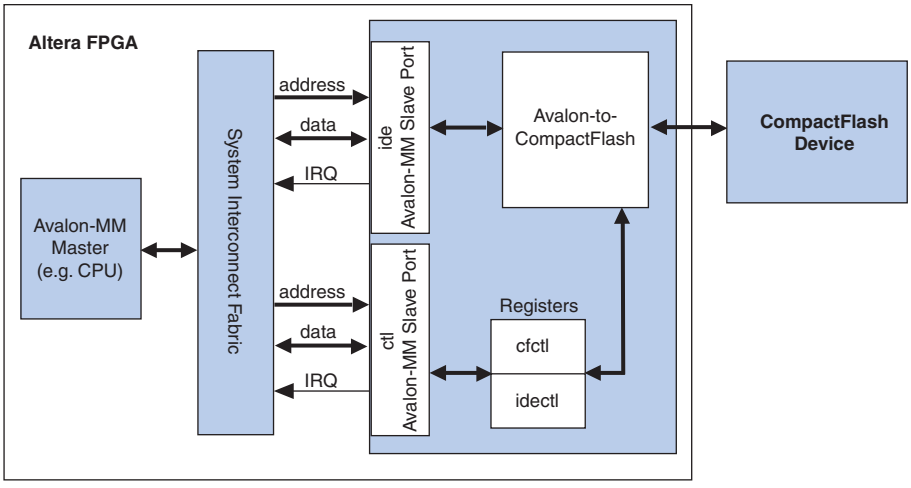
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 2–2
- "Device Support" on page 2–3
- "Software Programming Model" on page 2–3

Functional Description

Figure 2–1 shows a block diagram of the CompactFlash core in a typical system configuration.

Figure 2–1. SOPC Builder System With a CompactFlash Core



As shown in Figure 2–1, the CompactFlash core provides two Avalon-MM slave interfaces: the `ide` slave port for accessing the registers on the CompactFlash device and the `ctl` slave port for accessing the core's internal registers. These registers can be used by Avalon-MM master peripherals such as a Nios II processor to control the operations of the CompactFlash core and to transfer data to and from the CompactFlash device.

You can set the CompactFlash core to generate two active-high interrupt requests (IRQs): one signals the insertion and removal of a CompactFlash device and the other passes interrupt signals from the CompactFlash device.

The CompactFlash core maps the Avalon-MM bus signals to the CompactFlash device with proper timing, thus allowing Avalon-MM master peripherals to directly access the registers on the CompactFlash device.

For more information, refer to the CF+ and CompactFlash specifications available at www.compactflash.org.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the CompactFlash core in SOPC Builder to add the core to a system. There are no user-configurable settings for this core.

Required Connections

Table 2–1 lists the required connections between the CompactFlash core and the CompactFlash device.

Table 2–1. Required Connections (Part 1 of 2)

CompactFlash Interface Signal Name	Pin Type	CompactFlash Pin Number
`addr[0]`	Output	20
`addr[1]`	Output	19
`addr[2]`	Output	18
`addr[3]`	Output	17
`addr[4]`	Output	16
`addr[5]`	Output	15
`addr[6]`	Output	14
`addr[7]`	Output	12
`addr[8]`	Output	11
`addr[9]`	Output	10
`addr[10]`	Output	8
`atasel_n`	Output	9
`cs_n[0]`	Output	7
`cs_n[1]`	Output	32
`data[0]`	Input/Output	21
`data[1]`	Input/Output	22

Table 2–1. Required Connections (Part 2 of 2)

CompactFlash Interface Signal Name	Pin Type	CompactFlash Pin Number
data[2]	Input/Output	23
data[3]	Input/Output	2
data[4]	Input/Output	3
data[5]	Input/Output	4
data[6]	Input/Output	5
data[7]	Input/Output	6
data[8]	Input/Output	47
data[9]	Input/Output	48
data[10]	Input/Output	49
data[11]	Input/Output	27
data[12]	Input/Output	28
data[13]	Input/Output	29
data[14]	Input/Output	30
data[15]	Input/Output	31
detect	Input	25 or 26
intrq	Input	37
iord_n	Output	34
iordy	Input	42
iowr_n	Output	35
power	Output	CompactFlash power controller, if present
reset_n	Output	41
rfu	Output	44
we_n	Output	46

Device Support

The CompactFlash interface core supports all Altera® device families.

Software Programming Model

This section describes the software programming model for the CompactFlash core.

HAL System Library Support

The Altera-provided HAL API functions include a device driver that you can use to initialize the CompactFlash core. To perform other operations, use the low-level macros provided. For more information on the macros, refer to the files listed in the section "Software Files" on page 2–4.

Software Files

The CompactFlash core provides the following software files. These files define the low-level access to the hardware. Application developers should not modify these files.

- **altera_avalon_cf_regs.h**—The header file that defines the core's register maps.
- **altera_avalon_cf.h, altera_avalon_cf.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.

Register Maps

This section describes the register maps for the Avalon-MM slave interfaces.

Ide Registers

The `ide` port in the CompactFlash core allows you to access the IDE registers on a CompactFlash device. Table 2–2 shows the register map for the `ide` port.

Table 2–2. Ide Register Map

Offset	Register Names	
	Read Operation	**Write Operation**
0	`RD Data`	`WR Data`
1	`Error`	`Features`
2	`Sector Count`	`Sector Count`
3	`Sector No`	`Sector No`
4	`Cylinder Low`	`Cylinder Low`
5	`Cylinder High`	`Cylinder High`
6	`Select Card/Head`	`Select Card/Head`
7	`Status`	`Command`
14	`Alt Status`	`Device Control`

Ctl Registers

The `ctl` port in the CompactFlash core provides access to the registers which control the core's operation and interface. Table 2–3 shows the register map for the `ctl` port.

Table 2–3. Ctl Register Map

Offset	Register	Fields				
		31:4	**3**	**2**	**1**	**0**
0	`cfctl`	Reserved	`IDET`	`RST`	`PWR`	`DET`
1	`idectl`	Reserved				`IIDE`
2	Reserved	Reserved				
3	Reserved	Reserved				

Cfctl Register

The `cfctl` register controls the operations of the CompactFlash core. Reading the `cfctl` register clears the interrupt. Table 2–4 describes the `cfctl` register bits.

Table 2–4. cfctl Register Bits

Bit Number	Bit Name	Read/Write	Description
0	`DET`	RO	Detect. This bit is set to 1 when the core detects a CompactFlash device.
1	`PWR`	RW	Power. When this bit is set to 1, power is being supplied to the CompactFlash device.
2	`RST`	RW	Reset. When this bit is set to 1, the CompactFlash device is held in a reset state. Setting this bit to 0 returns the device to its active state.
3	`IDET`	RW	Detect Interrupt Enable. When this bit is set to 1, the CompactFlash core generates an interrupt each time the value of the `det` bit changes.

Idectl Register

The `idectl` register controls the interface to the CompactFlash device. Table 2–5 describes the `idectl` register bit.

Table 2–5. idectl Register

Bit Number	Bit Name	Read/Write	Description
0	`IIDE`	RW	IDE Interrupt Enable. When this bit is set to 1, the CompactFlash core generates an interrupt following an interrupt generated by the CompactFlash device. Setting this bit to 0 disables the IDE interrupt.

Document Revision History

Table 2–6 shows the revision history for this chapter.

Table 2–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Added the mode supported by the CompactFlash core.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

3. Common Flash Interface Controller Core

Core Overview

The common flash interface controller core with Avalon® interface (CFI controller) allows you to easily connect SOPC Builder systems to external flash memory that complies with the Common Flash Interface (CFI) specification. The CFI controller is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

For the Nios® II processor, Altera provides hardware abstraction layer (HAL) driver routines for the CFI controller. The drivers provide universal access routines for CFI-compliant flash memories. Therefore, you do not need to write any additional code to program CFI-compliant flash devices. The HAL driver routines take advantage of the HAL generic device model for flash memory, which allows you to access the flash memory using the familiar HAL application programming interface (API), the ANSI C standard library functions for file I/O, or both.

The Nios II Embedded Design Suite (EDS) provides a flash programmer utility based on the Nios II processor and the CFI controller. The flash programmer utility can be used to program any CFI-compliant flash memory connected to an Altera® device.

For more information about how to read and write flash using the HAL API, refer to the *Nios II Software Developer's Handbook*. For more information on the flash programmer utility, refer to the *Nios II Flash Programmer User Guide*.

Further information about the Common Flash Interface specification is available at www.intel.com. As an example of a flash device supported by the CFI controller, see the data sheet for the AMD Am29LV065D-120R, available at www.amd.com.

The common flash interface controller core supersedes previous Altera flash cores distributed with SOPC Builder or Nios development kits. All flash chips associated with these previous cores comply with the CFI specification, and therefore are supported by the CFI controller.

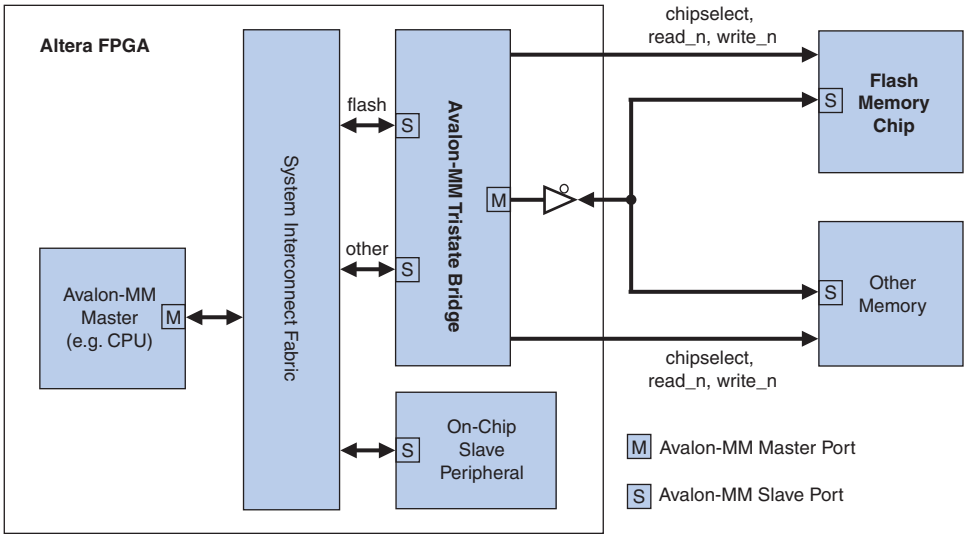
This chapter contains the following sections:

- "Functional Description" on page 3–2
- "Device and Tools Support" on page 3–2
- "Instantiating the Core in SOPC Builder" on page 3–2
- "Software Programming Model" on page 3–4

Functional Description

Figure 3–1 shows a block diagram of the CFI controller in a typical system configuration. As shown in Figure 3–1, the Avalon Memory-Mapped (Avalon-MM) interface for flash devices is connected through an Avalon-MM tristate bridge. The tristate bridge creates an off-chip memory bus that allows the flash chip to share address and data pins with other memory chips. It provides separate chipselect, read, and write pins to each chip connected to the memory bus. The CFI controller hardware is minimal; it is simply an Avalon-MM tristate slave port configured with waitstates, setup, and hold time appropriate for the target flash chip. This slave port is capable of Avalon-MM tristate slave read and write transfers.

Figure 3–1. An SOPC Builder System Integrating a CFI Controller



Avalon-MM master ports can perform read transfers directly from the CFI controller's Avalon-MM port. See "Software Programming Model" on page 3–4 for more detail on writing/erasing flash memory.

Device and Tools Support

The CFI controller supports all Altera device families. The CFI controller provides drivers for the Nios II HAL system library.

Instantiating the Core in SOPC Builder

Hardware designers use the MegaWizard™ interface for the CFI controller in SOPC Builder to specify the core features. The following sections describe the available options.

Attributes Page

The options on this page control the basic hardware configuration of the CFI controller.

Presets Settings

The **Presets** setting is a drop-down menu of flash chips that have already been characterized for use with the CFI controller. After you select one of the chips in the **Presets** menu, the wizard updates all settings on both tabs (except for the Board Info setting) to work with the specified flash chip.

The options provided are not intended to cover the wide range of flash devices available in the market. If the flash chip on your target board does not appear in the **Presets** list, you must configure the other settings manually.

Size Settings

The size setting specifies the size of the flash device. There are two settings:

- **Address Width**—The width of the flash chip's address bus.
- **Data Width—**The width of the flash chip's data bus

The size settings cause SOPC Builder to allocate the correct amount of address space for this device. SOPC Builder will automatically generate dynamic bus sizing logic that appropriately connects the flash chip to Avalon-MM master ports of different data widths.

For details about dynamic bus sizing, refer to the *Avalon Interface Specifications*.

Timing Page

The options on this page specify the timing requirements for read and write transfers with the flash device.

Refer to the specifications provided with the common flash device you are using to obtain the timing values you need to calculate the values of the parameters on the **Timing** page.

The settings available on the **Timing** page are:

- **Setup**—After asserting `chipselect`, the time required before asserting the `read` or `write` signals. You can determine the value of this parameter by using the following formula:

  Setup = t_{CE} (chip enable to output delay) - t_{OE} (output enable to output delay)

- **Wait**—The time required for the `read` or `write` signal to be asserted for each transfer. Use the following guideline to determine an appropriate value for this parameter:

  The sum of **Setup**, **Wait**, and board delay must be greater than t_{ACC}, where:

  - Board delay is determined by the T_{CO} on the FPGA address pins, T_{SU} on the device data pins, and propagation delay on the board traces in both directions.
  - t_{ACC} is the address to output delay.

- **Hold**—After deasserting the `write` signal, the time required before deasserting the `chipselect` signal.
- **Units**—The timing units used for the **Setup**, **Wait**, and **Hold** values. Possible values include ns, µs, ms, and clock cycles.

For more information about signal timing for the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

Software Programming Model

This section describes the software programming model for the CFI controller. In general, any Avalon-MM master in the system can read the flash chip directly as a memory device. For Nios II processor users, Altera provides HAL system library drivers that enable you to erase and write the flash memory using the HAL API functions.

HAL System Library Support

The Altera-provided driver implements a HAL flash device driver that integrates into the HAL system library for Nios II systems. Programs call the familiar HAL API functions to program CFI-compliant flash memory. You do not need to know anything about the details of the underlying drivers.

The HAL API for programming flash, including C code examples, is described in detail in the *Nios II Software Developer's Handbook*. The Nios II EDS also provides a reference design called Flash Tests that demonstrates erasing, writing, and reading flash memory.

Limitations

Currently, the Altera-provided drivers for the CFI controller support only Intel, AMD and Spansion flash chips.

Software Files

The CFI controller provides the following software files. These files define the low-level access to the hardware, and provide the routines for the HAL flash device driver. Application developers should not modify these files.

- **altera_avalon_cfi_flash.h, altera_avalon_cfi_flash.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.
- **altera_avalon_cfi_flash_funcs.h**, **altera_avalon_cfi_flash_table.c**—The header and source code for functions concerned with accessing the CFI table.
- **altera_avalon_cfi_flash_amd_funcs.h**, **altera_avalon_cfi_flash_amd.c**—The header and source code for programming AMD CFI-compliant flash chips.
- **altera_avalon_cfi_flash_intel_funcs.h**, **altera_avalon_cfi_flash_intel.c**—The header and source code for programming Intel CFI-compliant flash chips.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Nios II Flash Programmer User Guide*
- *Nios II Software Developer's Handbook*

Document Revision History

Table 3–1 shows the revision history for this chapter.

Table 3–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	Revised description of the timing page settings.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Added description to parameters on **Timing** page.	—
May 2008 v8.0.0	Updated the CFI controllers supported by Altera-provided drivers.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

4. EPCS Serial Flash Controller Core

Core Overview

The EPCS serial flash controller core with Avalon® interface allows Nios® II systems to access an Altera® EPCS serial configuration device. Altera provides drivers that integrate into the Nios II hardware abstraction layer (HAL) system library, allowing you to read and write the EPCS device using the familiar HAL application program interface (API) for flash devices.

Using the EPCS serial flash controller core, Nios II systems can:

- Store program code in the EPCS device. The EPCS serial flash controller core provides a boot-loader feature that allows Nios II systems to store the main program code in an EPCS device.
- Store non-volatile program data, such as a serial number, a NIC number, and other persistent data.
- Manage the device configuration data. For example, a network-enabled embedded system can receive new FPGA configuration data over a network, and use the core to program the new data into an EPCS serial configuration device.

The EPCS serial flash controller core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. The flash programmer utility in the Nios II IDE allows you to manage and program data contents into the EPCS device.

For information about the EPCS serial configuration device family, refer to the *Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64 and EPCS128) Data Sheet*. For details about using the Nios II HAL API to read and write flash memory, refer to the *Nios II Software Developer's Handbook*. For details about managing and programming the EPCS memory contents, refer to the *Nios II Flash Programmer User Guide*.

For Nios II processor users, the EPCS serial flash controller core supersedes the Active Serial Memory Interface (ASMI) device. New designs should use the EPCS serial flash controller core instead of the ASMI core.

This chapter contains the following sections:

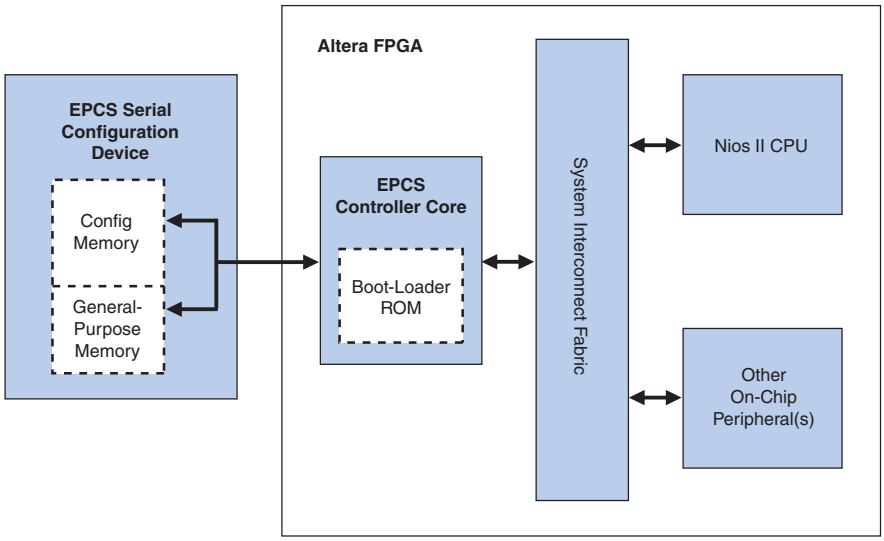
- "Functional Description" on page 4–2
- "Device and Tools Support" on page 4–4
- "Instantiating the Core in SOPC Builder" on page 4–4
- "Software Programming Model" on page 4–4

Functional Description

Figure 4–1 shows a block diagram of the EPCS serial flash controller core in a typical system configuration. As shown in Figure 4–1, the EPCS device's memory can be thought of as two separate regions:

- **FPGA configuration memory**—FPGA configuration data is stored in this region.
- **General-purpose memory**—If the FPGA configuration data does not fill up the entire EPCS device, any left-over space can be used for general-purpose data and system startup code.

Figure 4–1. Nios II System Integrating an EPCS Serial Flash Controller Core



By virtue of the HAL generic device model for flash devices, accessing the EPCS device using the HAL API is the same as accessing any flash memory. The EPCS device has a special-purpose hardware interface, so Nios II programs must read and write the EPCS memory using the provided HAL flash drivers.

The EPCS serial flash controller core contains an on-chip memory for storing a boot-loader program. When used in conjunction with Cyclone® and Cyclone II devices, the core requires 512 bytes of boot-loader ROM. For Cyclone III, Cyclone IV, Stratix® II, and newer device families in the Stratix series, the core requires 1 KByte of boot-loader ROM. The Nios II processor can be configured to boot from the EPCS serial flash controller core. To do so, set the Nios II reset address to the base address of the EPCS serial flash controller core. In this case, after reset the CPU first executes code from the boot-loader ROM, which copies data from the EPCS general-purpose memory region into a RAM. Then, program control transfers to the RAM. The Nios II IDE provides facilities to compile a program for storage in the EPCS device, and create a programming file to program into the EPCS device.

For more information, refer to the *Nios II Flash Programmer User Guide*.

If you program the EPCS device using the Quartus® II Programmer, all previous content is erased. To program the EPCS device with a combination of FPGA configuration data and Nios II program data, use the Nios II IDE flash programmer utility.

The Altera EPCS configuration device connects to the FPGA through dedicated pins on the FPGA, not through general-purpose I/O pins. In all Altera device families except Cyclone III and Cyclone IV, the EPCS serial flash controller core does not create any I/O ports on the top-level SOPC Builder system module. If the EPCS device and the FPGA are wired together on a board for configuration using the EPCS device (in other words, active serial configuration mode), no further connection is necessary between the EPCS serial flash controller core and the EPCS device. When you compile the SOPC Builder system in the Quartus II software, the EPCS serial flash controller core signals are routed automatically to the device pins for the EPCS device.

You, however, have the option not to use the dedicated pins on the FPGA (active serial configuration mode) by turning off the respective parameters in the MegaWizard interface. When this option is turned off or when the target device is a Cyclone III or Cyclone IV device, you have the flexibility to connect the output pins, which are exported to the top-level design, to any EPCS devices. Perform the following tasks in the Quartus® II software to make the necessary pin assignments:

- On the **Dual-purpose pins** page (**Assignments** > **Devices** > **Device and Pin Options**), ensure that the following pins are assigned to the respective values:
  - `Data[0]` = **Use as regular I/O**
  - `Data[1]` = **Use as regularr I/O**
  - `DCLK` = **Use as regular I/O**
  - `FLASH_nCE/nCS0` = **Use as regular I/O**
- Using the Pin Planner (**Assignments > Pins**), ensure that the following pins are assigned to the respective configuration functions on the device:
  - `data0_to_the_epcs_controller` = `DATA0`
  - `sdo_from the_epcs_controller` = `DATA1,ASDO`
  - `dclk_from_epcs_controller` = `DCLK`
  - `sce_from_the_epcs_controller` = `FLASH_nCE`

For more information about the configuration pins in Altera devices, refer to the Pin-Out Files for Altera Device page.

Avalon-MM Slave Interface and Registers

The EPCS serial flash controller core has a single Avalon-MM slave interface that provides access to both boot-loader code and registers that control the core. As shown in Table 4–1, the first segment is dedicated to the boot-loader code, and the next seven words are control and data registers. A Nios II CPU can read the instruction words, starting from the core's base address as flat memory space, which enables the CPU to reset the core's address space.

The EPCS serial flash controller core includes an interrupt signal that can be used to interrupt the CPU when a transfer has completed.

Table 4–1. EPCS Serial Flash Controller Core Register Map

Offset—Cyclone and Cyclone II (32-bit Word Address)	Offset—Other Device Families (32-bit Word Address)	Register Name	R/W	Bit Description
				31:0
0x00 .. 0x7F	0x00 .. 0xFF	Boot ROM Memory	R	Boot Loader Code
0x080	0x100	Read Data	R	(1)
0x081	0x101	Write Data	W	
0x082	0x102	Status	R/W	
0x083	0x103	Control	R/W	
0x084	0x104	Reserved	—	
0x085	0x105	Slave Enable	R/W	
0x086	0x106	End of Packet	R/W	

Note to Table 4–1:

(1) Altera does not publish the usage of the control and data registers. To access the EPCS device, you must use the HAL drivers provided by Altera.

Device and Tools Support

The EPCS serial flash controller core supports all Altera device families except the Hardcopy® series. The core must be connected to a Nios II processor. The core provides drivers for HAL-based Nios II systems, and the precompiled boot loader code compatible with the Nios II processor.

Instantiating the Core in SOPC Builder

You can add the EPCS serial flash controller core from the **System Contents** tab in SOPC Builder. In device families other than Cyclone III and Cyclone IV, you can use the MegaWizard™ interface to configure the core to use general I/O pins instead of dedicated pins by turning off both parameters, **Automatically select dedicated active serial interface, if supported** and **Use dedicated active serial interface**.

☞ Only one EPCS serial flash controller core can be instantiated in each FPGA design.

Software Programming Model

This section describes the software programming model for the EPCS serial flash controller core. Altera provides HAL system library drivers that enable you to erase and write the EPCS memory using the HAL API functions. Altera does not publish the usage of the cores registers. Therefore, you must use the HAL drivers provided by Altera to access the EPCS device.

HAL System Library Support

The Altera-provided driver implements a HAL flash device driver that integrates into the HAL system library for Nios II systems. Programs call the familiar HAL API functions to program the EPCS memory. You do not need to know the details of the underlying drivers to use them.

The driver for the EPCS device is excluded when the reduced device drivers option is enabled in a BSP or system library. To force inclusion of the EPCS drivers in a BSP with the reduced device drivers option enabled, you can define the preprocessor symbol, `ALT_USE_EPCS_FLASH`, before including the header, as follows:

```
#define ALT_USE_EPCS_FLASH
#include <altera_avalon_epcs_flash_controller.h>
```

The HAL API for programming flash, including C-code examples, is described in detail in the *Nios II Software Developer's Handbook*. For details about managing and programming the EPCS device contents, refer to the *Nios II Flash Programmer User Guide*.

Software Files

The EPCS serial flash controller core provides the following software files. These files provide low-level access to the hardware and drivers that integrate into the Nios II HAL system library. Application developers should not modify these files.

- **altera_avalon_epcs_flash_controller.h**, **altera_avalon_epcs_flash_controller.c**—Header and source files that define the drivers required for integration into the HAL system library.
- **epcs_commands.h**, **epcs_commands.c**—Header and source files that directly control the EPCS device hardware to read and write the device. These files also rely on the Altera SPI core drivers.

Referenced Documents

This chapter references the following documents:

- *Nios II Flash Programmer User Guide*
- *Nios II Software Developer's Handbook*
- *Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64 and EPCS128) Data Sheet*

Document Revision History

Table 4–2 shows the revision history for this chapter.

Table 4–2. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	■ Revised descriptions of register fields and bits. ■ Updated the section on HAL System Library Support.	—
March 2009 v9.0.0	Updated the boot ROM memory offset for other device familes in Table 4–1.	—

Table 4–2. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	■ Updated the boot rom size. ■ Added additional steps to perform to connect output pins in Cyclone III devices.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

5. JTAG UART Core

Core Overview

The JTAG UART core with Avalon® interface implements a method to communicate serial character streams between a host PC and an SOPC Builder system on an Altera® FPGA. In many designs, the JTAG UART core eliminates the need for a separate RS-232 serial connection to a host PC for character I/O. The core provides an Avalon interface that hides the complexities of the JTAG interface from embedded software programmers. Master peripherals (such as a Nios® II processor) communicate with the core by reading and writing control and data registers.

The JTAG UART core uses the JTAG circuitry built in to Altera FPGAs, and provides host access via the JTAG pins on the FPGA. The host PC can connect to the FPGA via any Altera JTAG download cable, such as the USB-Blaster™ cable. Software support for the JTAG UART core is provided by Altera. For the Nios II processor, device drivers are provided in the HAL system library, allowing software to access the core using the ANSI C Standard Library **stdio.h** routines. For the host PC, Altera provides JTAG terminal software that manages the connection to the target, decodes the JTAG data stream, and displays characters on screen.

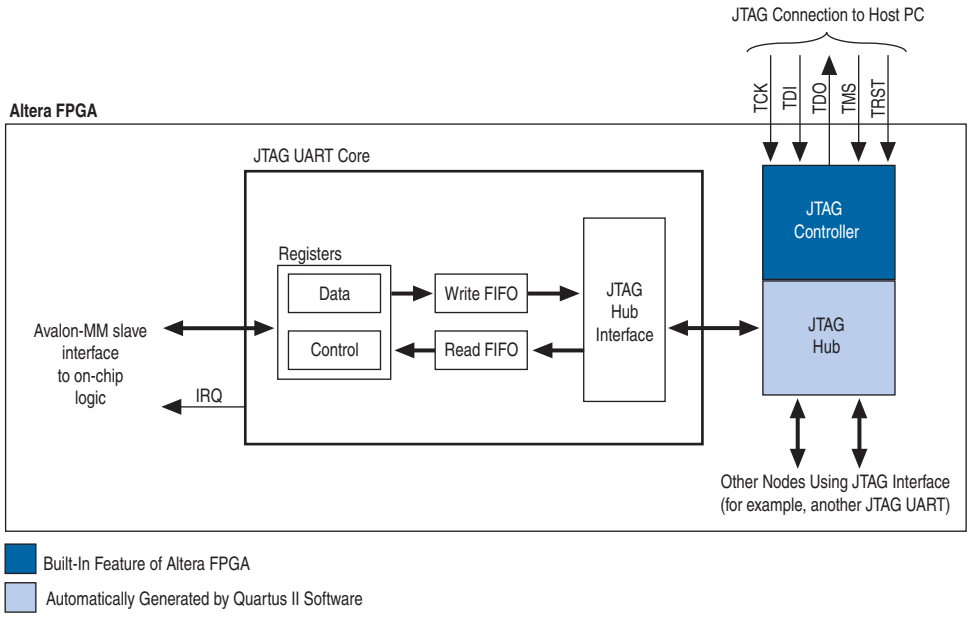
The JTAG UART core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 5–2
- "Device and Tools Support" on page 5–4
- "Instantiating the Core in SOPC Builder" on page 5–4
- "Hardware Simulation Considerations" on page 5–6
- "Software Programming Model" on page 5–6

Functional Description

Figure 5–1 shows a block diagram of the JTAG UART core and its connection to the JTAG circuitry inside an Altera FPGA. The following sections describe the components of the core.

Figure 5–1. JTAG UART Core Block Diagram



Avalon Slave Interface and Registers

The JTAG UART core provides an Avalon slave interface to the JTAG circuitry on an Altera FPGA. The user-visible interface to the JTAG UART core consists of two 32-bit registers, `data` and `control`, that are accessed through an Avalon slave port. An Avalon master, such as a Nios II processor, accesses the registers to control the core and transfer data over the JTAG connection. The core operates on 8-bit units of data at a time; eight bits of the `data` register serve as a one-character payload.

The JTAG UART core provides an active-high interrupt output that can request an interrupt when read data is available, or when the write FIFO is ready for data. For further details see "Interrupt Behavior" on page 5–11.

Read and Write FIFOs

The JTAG UART core provides bidirectional FIFOs to improve bandwidth over the JTAG connection. The FIFO depth is parameterizable to accommodate the available on-chip memory. The FIFOs can be constructed out of memory blocks or registers, allowing you to trade off logic resources for memory resources, if necessary.

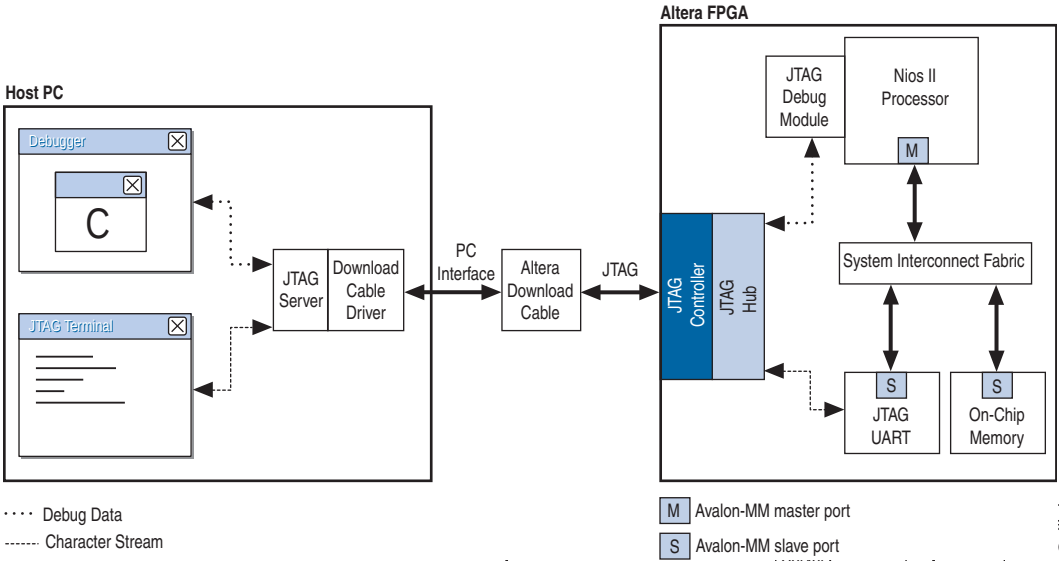
JTAG Interface

Altera FPGAs contain built-in JTAG control circuitry between the device's JTAG pins and the logic inside the device. The JTAG controller can connect to user-defined circuits called *nodes* implemented in the FPGA. Because several nodes may need to communicate via the JTAG interface, a JTAG hub, which is a multiplexer, is necessary. During logic synthesis and fitting, the Quartus® II software automatically generates the JTAG hub logic. No manual design effort is required to connect the JTAG circuitry inside the device; the process is presented here only for clarity.

Host-Target Connection

Figure 5–2 shows the connection between a host PC and an SOPC Builder-generated system containing a JTAG UART core.

Figure 5–2. Example System Using the JTAG UART Core



The JTAG controller on the FPGA and the download cable driver on the host PC implement a simple data-link layer between host and target. All JTAG nodes inside the FPGA are multiplexed through the single JTAG connection. JTAG server software on the host PC controls and decodes the JTAG data stream, and maintains distinct connections with nodes inside the FPGA.

The example system in Figure 5–2 contains one JTAG UART core and a Nios II processor. Both agents communicate with the host PC over a single Altera download cable. Thanks to the JTAG server software, each host application has an independent connection to the target. Altera provides the JTAG server drivers and host software required to communicate with the JTAG UART core.

☞ Systems with multiple JTAG UART cores are possible, and all cores communicate via the same JTAG interface. To maintain coherent data streams, only one processor should communicate with each JTAG UART core.

Device and Tools Support

The JTAG UART core supports all Altera® device families. The JTAG UART core is supported by the Nios II hardware abstraction layer (HAL) system library.

To view the character stream on the host PC, the JTAG UART core must be used in conjunction with the JTAG terminal software provided by Altera. Nios II processor users access the JTAG UART via the Nios II IDE or the **nios2-terminal** command-line utility.

For further details, refer to the *Nios II Software Developer's Handbook* or the Nios II IDE online help.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the JTAG UART core in SOPC Builder to specify the core features. The following sections describe the available options.

Configuration Page

The options on this page control the hardware configuration of the JTAG UART core. The default settings are pre-configured to behave optimally with the Altera-provided device drivers and JTAG terminal software. Most designers should not change the default values, except for the **Construct using registers instead of memory blocks** option.

Write FIFO Settings

The write FIFO buffers data flowing from the Avalon interface to the host. The following settings are available:

- **Depth**—The write FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.
- **IRQ Threshold—**The write IRQ threshold governs how the core asserts its IRQ in response to the FIFO emptying. As the JTAG circuitry empties data from the write FIFO, the core asserts its IRQ when the number of characters remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by writing more data and preventing the write FIFO from emptying completely. A value of 8 is typically optimal. See "Interrupt Behavior" on page 5–11 for further details.
- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of on-chip logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 logic elements (LEs), so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.

Read FIFO Settings

The read FIFO buffers data flowing from the host to the Avalon interface. Settings are available to control the depth of the FIFO and the generation of interrupts.

- **Depth**—The read FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.
- **IRQ Threshold—**The IRQ threshold governs how the core asserts its IRQ in response to the FIFO filling up. As the JTAG circuitry fills up the read FIFO, the core asserts its IRQ when the amount of space remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by reading data and preventing the read FIFO from filling up completely. A value of 8 is typically optimal. See "Interrupt Behavior" on page 5–11 for further details.
- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 LEs, so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.

Simulation Settings

At system generation time, when SOPC Builder generates the logic for the JTAG UART core, a simulation model is also constructed. The simulation model offers features to simplify simulation of systems using the JTAG UART core. Changes to the simulation settings do not affect the behavior of the core in hardware; the settings affect only functional simulation.

Simulated Input Character Stream

You can enter a character stream that will be simulated entering the read FIFO upon simulated system reset. The MegaWizard Interface accepts an arbitrary character string, which is later incorporated into the test bench. After reset, this character string is pre-initialized in the read FIFO, giving the appearance that an external JTAG terminal program is sending a character stream to the JTAG UART core.

Prepare Interactive Windows

At system generation time, the JTAG UART core generator can create ModelSim® macros to open interactive windows during simulation. These windows allow the user to send and receive ASCII characters via a console, giving the appearance of a terminal session with the system executing in hardware. The following options are available:

- **Do not generate ModelSim aliases for interactive windows**—This option does not create any ModelSim macros for character I/O.
- **Create ModelSim alias to open a window showing output as ASCII text**—This option creates a ModelSim macro to open a console window that displays output from the write FIFO. Values written to the write FIFO via the Avalon interface are displayed in the console as ASCII characters.

- **Create ModelSim alias to open an interactive stimulus/response window**—This option creates a ModelSim macro to open a console window that allows input and output interaction with the core. Values written to the write FIFO via the Avalon interface are displayed in the console as ASCII characters. Characters typed into the console are fed into the read FIFO, and can be read via the Avalon interface. When this option is enabled, the simulated character input stream option is ignored.

Hardware Simulation Considerations

The simulation features were created for easy simulation of Nios II processor systems when using the ModelSim simulator. The simulation model is implemented in the JTAG UART core's top-level HDL file. The synthesizable HDL and the simulation HDL are implemented in the same file. Some simulation features are implemented using `translate on/off` synthesis directives that make certain sections of HDL code visible only to the synthesis tool.

For complete details about simulating the JTAG UART core in Nios II systems, refer to *AN 351: Simulating Nios II Processor Designs*.

Other simulators can be used, but require user effort to create a custom simulation process. You can use the auto-generated ModelSim scripts as references to create similar functionality for other simulators.

CAUTION Do not edit the simulation directives if you are using Altera's recommended simulation procedures. If you change the simulation directives to create a custom simulation flow, be aware that SOPC Builder overwrites existing files during system generation. Take precautions to ensure your changes are not overwritten.

Software Programming Model

The following sections describe the software programming model for the JTAG UART core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the JTAG UART using the ANSI C standard library functions, such as `printf()` and `getchar()`.

HAL System Library Support

The Altera-provided driver implements a HAL character-mode device driver that integrates into the HAL system library for Nios II systems. HAL users should access the JTAG UART via the familiar HAL API and the ANSI C standard library, rather than accessing the JTAG UART registers. `ioctl()` requests are defined that allow HAL users to control the hardware-dependent aspects of the JTAG UART.

CAUTION If your program uses the Altera-provided HAL device driver to access the JTAG UART hardware, accessing the device registers directly will interfere with the correct behavior of the driver.

For Nios II processor users, the HAL system library API provides complete access to the JTAG UART core's features. Nios II programs treat the JTAG UART core as a character mode device, and send and receive data using the ANSI C standard library functions, such as `getchar()` and `printf()`.

Example 5–1 demonstrates the simplest possible usage, printing a message to `stdout` using `printf()`. In this example, the SOPC Builder system contains a JTAG UART core, and the HAL system library is configured to use this JTAG UART device for `stdout`.

Example 5–1. *Printing Characters to a JTAG UART Core as stdout*

```
#include <stdio.h>
int main ()
{
   printf("Hello world.\n");
   return 0;
}
```

Example 5–2 demonstrates reading characters from and sending messages to a JTAG UART core using the C standard library. In this example, the SOPC Builder system contains a JTAG UART core named `jtag_uart` that is not necessarily configured as the `stdout` device. In this case, the program treats the device like any other node in the HAL file system.

Example 5–2. Transmitting Characters to a JTAG UART Core

```
/* A simple program that recognizes the characters 't' and 'v' */
#include <stdio.h>
#include <string.h>
int main ()
{
   char* msg = "Detected the character 't'.\n";
   FILE* fp;
   char prompt = 0;

fp = fopen ("/dev/jtag_uart", "r+"); //Open file for reading and writing
if (fp)
{
   while (prompt != 'v')
   {  // Loop until we receive a 'v'.
      prompt = getc(fp);  // Get a character from the JTAG UART.
      if (prompt == 't')
      {  // Print a message if character is 't'.
         fwrite (msg, strlen (msg), 1, fp);
      }

      if (ferror(fp))// Check if an error occurred with the file
      pointer clearerr(fp);// If so, clear it.
   }

   fprintf(fp, "Closing the JTAG UART file handle.\n");
   fclose (fp);
   }

   return 0;
}
```

In this example, the `ferror(fp)` is used to check if an error occurred on the JTAG UART connection, such as a disconnected JTAG connection. In this case, the driver detects that the JTAG connection is disconnected, reports an error (`EIO`), and discards data for subsequent transactions. If this error ever occurs, the C library latches the value until you explicitly clear it with the `clearerr()` function.

For complete details of the HAL system library, refer to the *Nios II Software Developer's Handbook*.

The Nios II Embedded Design Suite (EDS) provides a number of software example designs that use the JTAG UART core.

Driver Options: Fast vs. Small Implementations

To accommodate the requirements of different types of systems, the JTAG UART driver has two variants, a fast version and a small version. The fast behavior is used by default. Both the fast and small drivers fully support the C standard library functions and the HAL API.

The fast driver is an interrupt-driven implementation, which allows the processor to perform other tasks when the device is not ready to send or receive data. Because the JTAG UART data rate is slow compared to the processor, the fast driver can provide a large performance benefit for systems that could be performing other tasks in the interim. In addition, the fast version of the Altera Avalon JTAG UART monitors the connection to the host. The driver discards characters if no host is connected, or if the host is not running an application that handles the I/O stream.

The small driver is a polled implementation that waits for the JTAG UART hardware before sending and receiving each character. The performance of the small driver is poor if you are sending large amounts of data. The small version assumes that the host is always connected, and will never discard characters. Therefore, the small driver will hang the system if the JTAG UART hardware is ever disconnected from the host while the program is sending or receiving data. There are two ways to enable the small footprint driver:

- Enable the small footprint setting for the HAL system library project. This option affects device drivers for all devices in the system.
- Specify the preprocessor option `-DALTERA_AVALON_JTAG_UART_SMALL`. Use this option if you want the small, polled implementation of the JTAG UART driver, but you do not want to affect the drivers for other devices.

ioctl() Operations

The fast version of the JTAG UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. Specifically, you can use the `ioctl()` operations to control the timeout period, and to detect whether or not a host is connected. The fast driver defines the `ioctl()` operations shown in Table 5–1.

Table 5–1. JTAG UART ioctl() Operations for the Fast Driver Only

Request	Meaning
`TIOCSTIMEOUT`	Set the timeout (in seconds) after which the driver will decide that the host is not connected. A timeout of 0 makes the target assume that the host is always connected. The `ioctl` arg parameter passed in must be a pointer to an integer.
`TIOCGCONNECTED`	Sets the integer arg parameter to a value that indicates whether the host is connected and acting as a terminal (1), or not connected (0). The `ioctl` arg parameter passed in must be a pointer to an integer.

For details about the `ioctl()` function, refer to the *Nios II Software Developer's Handbook*.

Software Files

The JTAG UART core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_jtag_uart_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **altera_avalon_jtag_uart.h**, **altera_avalon_jtag_uart.c**—These files implement the HAL system library device driver.

Accessing the JTAG UART Core via a Host PC

Host software is necessary for a PC to access the JTAG UART core. The Nios II IDE supports the JTAG UART core, and displays character I/O in a console window. Altera also provides a command-line utility called **nios2-terminal** that opens a terminal session with the JTAG UART core.

For further details, refer to the *Nios II Software Developer's Handbook* and Nios II IDE online help.

Register Map

Programmers using the HAL API never access the JTAG UART core directly via its registers. In general, the register map is only useful to programmers writing a device driver for the core.

CAUTION

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 5–2 shows the register map for the JTAG UART core. Device drivers control and communicate with the core through the two, 32-bit memory-mapped registers.

Table 5–2. JTAG UART Core Register Map

Offset	Register Name	R/W	Bit Description														
			31	...	16	15	14	...	11	10	9	8	7	...	2	1	0
0	data	RW	RAVAIL			RVALID	Reserved						DATA				
1	control	RW	WSPACE			Reserved				AC	WI	RI	Reserved			WE	RE

Note to Table 5–2:

(1) Reserved fields—Read values are undefined. Write zero.

Data Register

Embedded software accesses the read and write FIFOs via the `data` register. Table 5–3 describes the function of each bit.

Table 5–3. data Register Bits

Bit(s)	Name	Access	Description
[7:0]	DATA	R/W	The value to transfer to/from the JTAG core. When writing, the DATA field holds a character to be written to the write FIFO. When reading, the DATA field holds a character read from the read FIFO.
[15]	RVALID	R	Indicates whether the DATA field is valid. If RVALID=1, the DATA field is valid, otherwise DATA is undefined.
[32:16]	RAVAIL	R	The number of characters remaining in the read FIFO (after the current read).

A read from the `data` register returns the first character from the FIFO (if one is available) in the DATA field. Reading also returns information about the number of characters remaining in the FIFO in the RAVAIL field. A write to the `data` register stores the value of the DATA field in the write FIFO. If the write FIFO is full, the character is lost.

Control Register

Embedded software controls the JTAG UART core's interrupt generation and reads status information via the `control` register. Table 5–4 describes the function of each bit.

Table 5–4. Control Register Bits

Bit(s)	Name	Access	Description
0	RE	R/W	Interrupt-enable bit for read interrupts.
1	WE	R/W	Interrupt-enable bit for write interrupts.
8	RI	R	Indicates that the read interrupt is pending.
9	WI	R	Indicates that the write interrupt is pending.
10	AC	R/C	Indicates that there has been JTAG activity since the bit was cleared. Writing 1 to AC clears it to 0.
[32:16]	WSPACE	R	The number of spaces available in the write FIFO.

A read from the `control` register returns the status of the read and write FIFOs. Writes to the register can be used to enable/disable interrupts, or clear the AC bit.

The RE and WE bits enable interrupts for the read and write FIFOs, respectively. The WI and RI bits indicate the status of the interrupt sources, qualified by the values of the interrupt enable bits (WE and RE). Embedded software can examine RI and WI to determine the condition that generated the IRQ. See "Interrupt Behavior" on page 5–11 for further details.

The AC bit indicates that an application on the host PC has polled the JTAG UART core via the JTAG interface. Once set, the AC bit remains set until it is explicitly cleared via the Avalon interface. Writing 1 to AC clears it. Embedded software can examine the AC bit to determine if a connection exists to a host PC. If no connection exists, the software may choose to ignore the JTAG data stream. When the host PC has no data to transfer, it can choose to poll the JTAG UART core as infrequently as once per second. Delays caused by other host software using the JTAG download cable could cause delays of up to 10 seconds between polls.

Interrupt Behavior

The JTAG UART core generates an interrupt when either of the individual interrupt conditions is pending and enabled.

☞ Interrupt behavior is of interest to device driver programmers concerned with the bandwidth performance to the host PC. Example designs and the JTAG terminal program provided with Nios II Embedded Design Suite (EDS) are pre-configured with optimal interrupt behavior.

The JTAG UART core has two kinds of interrupts: write interrupts and read interrupts. The WE and RE bits in the control register enable/disable the interrupts.

The core can assert a write interrupt whenever the write FIFO is nearly empty. The nearly empty threshold, write_threshold, is specified at system generation time and cannot be changed by embedded software. The write interrupt condition is set whenever there are write_threshold or fewer characters in the write FIFO. It is cleared by writing characters to fill the write FIFO beyond the write_threshold. Embedded software should only enable write interrupts after filling the write FIFO. If it has no characters remaining to send, embedded software should disable the write interrupt.

The core can assert a read interrupt whenever the read FIFO is nearly full. The nearly full threshold value, read_threshold, is specified at system generation time and cannot be changed by embedded software. The read interrupt condition is set whenever the read FIFO has read_threshold or fewer spaces remaining. The read interrupt condition is also set if there is at least one character in the read FIFO and no more characters are expected. The read interrupt is cleared by reading characters from the read FIFO.

For optimum performance, the interrupt thresholds should match the interrupt response time of the embedded software. For example, with a 10-MHz JTAG clock, a new character is provided (or consumed) by the host PC every 1 µs. With a threshold of 8, the interrupt response time must be less than 8 µs. If the interrupt response time is too long, performance suffers. If it is too short, interrupts occurs too often.

☞ For Nios II processor systems, read and write thresholds of 8 are an appropriate default.

Referenced Documents

This chapter references the *Nios II Software Developer's Handbook*.

Document Revision History

Table 5–5 shows the revision history for this chapter.

Table 5–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

Core Overview

The UART core with Avalon® interface implements a method to communicate serial character streams between an embedded system on an Altera® FPGA and an external device. The core implements the RS-232 protocol timing, and provides adjustable baud rate, parity, stop, and data bits, and optional `RTS/CTS` flow control signals. The feature set is configurable, allowing designers to implement just the necessary functionality for a given system.

The core provides an Avalon Memory-Mapped (Avalon-MM) slave interface that allows Avalon-MM master peripherals (such as a Nios® II processor) to communicate with the core simply by reading and writing control and data registers.

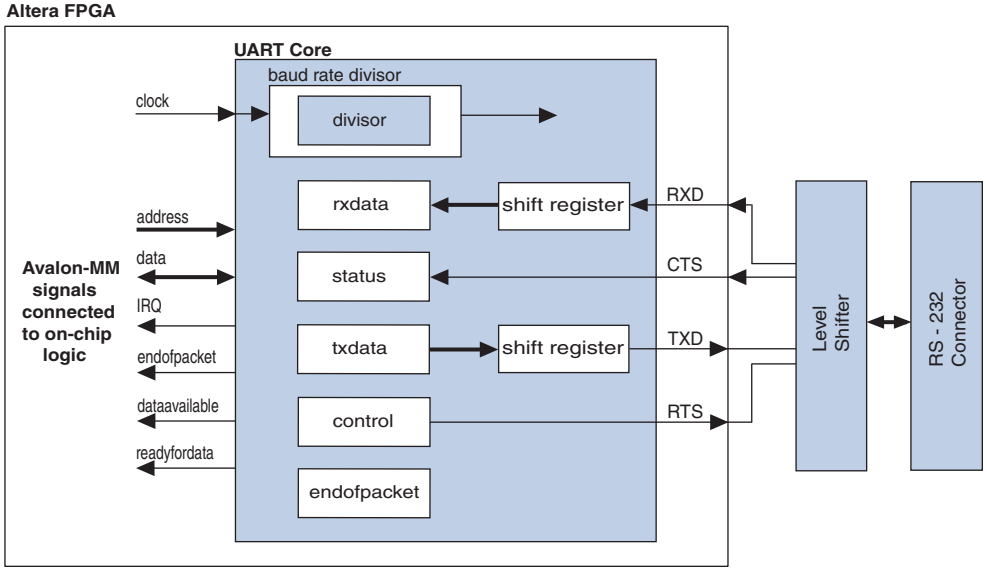
The UART core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 6–3
- "Instantiating the Core in SOPC Builder" on page 6–3
- "Simulation Considerations" on page 6–7
- "Software Programming Model" on page 6–8

Functional Description

Figure 6–1 shows a block diagram of the UART core.

Figure 6–1. Block Diagram of the UART Core in a Typical System



The core has two user-visible parts:

- The register file, which is accessed via the Avalon-MM slave port
- The RS-232 signals, `RXD`, `TXD`, `CTS`, and `RTS`

Avalon-MM Slave Interface and Registers

The UART core provides an Avalon-MM slave interface to the internal register file. The user interface to the UART core consists of six, 16-bit registers: `control`, `status`, `rxdata`, `txdata`, `divisor`, and `endofpacket`. A master peripheral, such as a Nios II processor, accesses the registers to control the core and transfer data over the serial connection.

The UART core provides an active-high interrupt request (IRQ) output that can request an interrupt when new data has been received, or when the core is ready to transmit another character. For further details, refer "Interrupt Behavior" on page 6–15.

The Avalon-MM slave port is capable of transfers with flow control. The UART core can be used in conjunction with a direct memory access (DMA) peripheral with Avalon-MM flow control to automate continuous data transfers between, for example, the UART core and memory.

For more information, refer to "Interval Timer Core" on page 27–1. For details about the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

RS-232 Interface

The UART core implements RS-232 asynchronous transmit and receive logic. The UART core sends and receives serial data via the `TXD` and `RXD` ports. The I/O buffers on most Altera FPGA families do not comply with RS-232 voltage levels, and may be damaged if driven directly by signals from an RS-232 connector. To comply with RS-232 voltage signaling specifications, an external level-shifting buffer is required (for example, Maxim MAX3237) between the FPGA I/O pins and the external RS-232 connector.

The UART core uses a logic 0 for mark, and a logic 1 for space. An inverter inside the FPGA can be used to reverse the polarity of any of the RS-232 signals, if necessary.

Transmitter Logic

The UART transmitter consists of a 7-, 8-, or 9-bit `txdata` holding register and a corresponding 7-, 8-, or 9-bit transmit shift register. Avalon-MM master peripherals write the `txdata` holding register via the Avalon-MM slave port. The transmit shift register is loaded from the `txdata` register automatically when a serial transmit shift operation is not currently in progress. The transmit shift register directly feeds the `TXD` output. Data is shifted out to `TXD` LSB first.

These two registers provide double buffering. A master peripheral can write a new value into the `txdata` register while the previously written character is being shifted out. The master peripheral can monitor the transmitter's status by reading the `status` register's transmitter ready (`TRDY`), transmitter shift register empty (`tmt`), and transmitter overrun error (`TOE`) bits.

The transmitter logic automatically inserts the correct number of start, stop, and parity bits in the serial `TXD` data stream as required by the RS-232 specification.

Receiver Logic

The UART receiver consists of a 7-, 8-, or 9-bit receiver-shift register and a corresponding 7-, 8-, or 9-bit `rxdata` holding register. Avalon-MM master peripherals read the `rxdata` holding register via the Avalon-MM slave port. The `rxdata` holding register is loaded from the receiver shift register automatically every time a new character is fully received.

These two registers provide double buffering. The `rxdata` register can hold a previously received character while the subsequent character is being shifted into the receiver shift register.

A master peripheral can monitor the receiver's status by reading the `status` register's read-ready (`RRDY`), receiver-overrun error (`ROE`), break detect (`BRK`), parity error (`PE`), and framing error (`FE`) bits. The receiver logic automatically detects the correct number of start, stop, and parity bits in the serial RXD stream as required by the RS-232 specification. The receiver logic checks for four exceptional conditions, frame error, parity error, receive overrun error, and break, in the received data and sets corresponding `status` register bits.

Baud Rate Generation

The UART core's internal baud clock is derived from the Avalon-MM clock input. The internal baud clock is generated by a clock divider. The divisor value can come from one of the following sources:

- A constant value specified at system generation time
- The 16-bit value stored in the `divisor` register

The `divisor` register is an optional hardware feature. If it is disabled at system generation time, the divisor value is fixed and the baud rate cannot be altered.

Device Support

The UART core supports all Altera® device families.

Instantiating the Core in SOPC Builder

Instantiating the UART in hardware creates at least two I/O ports for each UART core: An `RXD` input, and a `TXD` output. Optionally, the hardware may include flow control signals, the `CTS` input and `RTS` output.

Use the MegaWizard™ interface for the UART core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.

Configuration Settings

This section describes the configuration settings.

Baud Rate Options

The UART core can implement any of the standard baud rates for RS-232 connections. The baud rate can be configured in one of two ways:

- **Fixed rate**—The baud rate is fixed at system generation time and cannot be changed via the Avalon-MM slave port.
- **Variable rate**—The baud rate can vary, based on a clock divisor value held in the `divisor` register. A master peripheral changes the baud rate by writing new values to the `divisor` register.

☞ The baud rate is calculated based on the clock frequency provided by the Avalon-MM interface. Changing the system clock frequency in hardware without regenerating the UART core hardware results in incorrect signaling.

Baud Rate (bps) Setting

The **Baud Rate** setting determines the default baud rate after reset. The **Baud Rate** option offers standard preset values.

The baud rate value is used to calculate an appropriate clock divisor value to implement the desired baud rate. Baud rate and divisor values are related as shown in Equation 6–1 and Equation 6–2:

Equation 6–1.

$$\text{divisor} = \text{int}\left(\frac{\text{clock frequency}}{\text{baud rate}} + 0.5\right)$$

Equation 6–2.

$$\text{baud rate} = \frac{\text{clock frequency}}{\text{divisor} + 1}$$

Baud Rate Can Be Changed By Software Setting

When this setting is on, the hardware includes a 16-bit `divisor` register at address offset 4. The `divisor` register is writable, so the baud rate can be changed by writing a new value to this register.

When this setting is off, the UART hardware does not include a `divisor` register. The UART hardware implements a constant baud divisor, and the value cannot be changed after system generation. In this case, writing to address offset 4 has no effect, and reading from address offset 4 produces an undefined result.

Data Bits, Stop Bits, Parity

The UART core's parity, data bits and stop bits are configurable. These settings are fixed at system generation time; they cannot be altered via the register file. Table 6–1 explains the settings.

Table 6–1. Data Bits Settings

Setting	Legal Values	Description
Data Bits	7, 8, 9	This setting determines the widths of the `txdata`, `rxdata`, and `endofpacket` registers.
Stop Bits	1, 2	This setting determines whether the core transmits 1 or 2 stop bits with every character. The core always terminates a receive transaction at the first stop bit, and ignores all subsequent stop bits, regardless of this setting.
Parity	None, Even, Odd	This setting determines whether the UART core transmits characters with parity checking, and whether it expects received characters to have parity checking. When **Parity** is set to **None**, the transmit logic sends data without including a parity bit, and the receive logic presumes the incoming data does not include a parity bit. The `PE` bit in the `status` register is not implemented; it always reads 0. When **Parity** is set to **Odd** or **Even**, the transmit logic computes and inserts the required parity bit into the outgoing TXD bitstream, and the receive logic checks the parity bit in the incoming RXD bitstream. If the receiver finds data with incorrect parity, the `PE` bit in the `status` register is set to 1. When **Parity** is **Even**, the parity bit is 0 if the character has an even number of 1 bits; otherwise the parity bit is 1. Similarly, when parity is **Odd**, the parity bit is 0 if the character has an odd number of 1 bits.

Synchronizer Stages

The option **Synchronizer Stages** allows you to specify the length of synchronization register chains. These register chains are used when a metastable event is likely to occur and the length specified determines the meantime before failure. The register chain length, however, affects the latency of the core.

For more information on metastability in Altera devices, refer to *AN 42: Metastability in Altera Devices*. For more information on metastability analysis and synchronization register chains, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Flow Control

When the option **Include CTS/RTS pins and control register bits** is turned on, the UART core includes the following features:

- `cts_n` (logic negative CTS) input port
- `rts_n` (logic negative RTS) output port
- `CTS` bit in the `status` register
- `DCTS` bit in the `status` register
- `RTS` bit in the `control` register
- `IDCTS` bit in the `control` register

Based on these hardware facilities, an Avalon-MM master peripheral can detect CTS and transmit RTS flow control signals. The `CTS` input and `RTS` output ports are tied directly to bits in the `status` and `control` registers, and have no direct effect on any other part of the core. When using flow control, be sure the terminal program on the host side is also configured for flow control.

When the **Include CTS/RTS pins and control register bits** setting is off, the core does not include the aforementioned hardware and continuous writes to the UART may loose data. The control/status bits `CTS`, `DCTS`, `IDCTS`, and `RTS` are not implemented; they always read as 0.

Streaming Data (DMA) Control

The UART core's Avalon-MM interface optionally implements Avalon-MM transfers with flow control. Flow control allows an Avalon-MM master peripheral to write data only when the UART core is ready to accept another character, and to read data only when the core has data available. The UART core can also optionally include the end-of-packet register.

Include End-of-Packet Register

When this setting is on, the UART core includes:

- A 7-, 8-, or 9-bit `endofpacket` register at address-offset 5. The data width is determined by the **Data Bits** setting.
- EOP bit in the `status` register.
- IEOP bit in the `control` register.
- `endofpacket` signal in the Avalon-MM interface to support data transfers with flow control to and from other master peripherals in the system.

End-of-packet (EOP) detection allows the UART core to terminate a data transaction with an Avalon-MM master with flow control. EOP detection can be used with a DMA controller, for example, to implement a UART that automatically writes received characters to memory until a specified character is encountered in the incoming RXD stream. The terminating (EOP) character's value is determined by the `endofpacket` register.

When the EOP register is disabled, the UART core does not include the EOP resources. Writing to the `endofpacket` register has no effect, and reading produces an undefined value.

Simulation Settings

When the UART core's logic is generated, a simulation model is also created. The simulation model offers features to simplify and accelerate simulation of systems that use the UART core. Changes to the simulation settings do not affect the behavior of the UART core in hardware; the settings affect only functional simulation.

For examples of how to use the following settings to simulate Nios II systems, refer to *AN 351: Simulating Nios II Embedded Processor Designs*.

Simulated RXD-Input Character Stream

You can enter a character stream that is simulated entering the `RXD` port upon simulated system reset. The UART core's MegaWizard™ interface accepts an arbitrary character string, which is later incorporated into the UART simulation model. After reset in reset, the string is input into the `RXD` port character-by-character as the core is able to accept new data.

Prepare Interactive Windows

At system generation time, the UART core generator can create ModelSim macros that facilitate interaction with the UART model during simulation. You can turn on the following options:

- **Create ModelSim alias to open streaming output window** to create a ModelSim macro that opens a window to display all output from the `TXD` port.
- **Create ModelSim alias to open interactive stimulus window** to create a ModelSim macro that opens a window to accept stimulus for the `RXD` port. The window sends any characters typed in the window to the `RXD` port.

Simulated Transmitter Baud Rate

RS-232 transmission rates are often slower than any other process in the system, and it is seldom useful to simulate the functional model at the true baud rate. For example, at 115,200 bps, it typically takes thousands of clock cycles to transfer a single character. The UART simulation model has the ability to run with a constant clock divisor of 2, allowing the simulated UART to transfer bits at half the system clock speed, or roughly one character per 20 clock cycles. You can choose one of the following options for the simulated transmitter baud rate:

- **Accelerated (use divisor = 2)**—`TXD` emits one bit per 2 clock cycles in simulation.
- **Actual (use true baud divisor)**—`TXD` transmits at the actual baud rate, as determined by the `divisor` register.

Simulation Considerations

The simulation features were created for easy simulation of Nios II processor systems when using the ModelSim simulator. The documentation for the processor documents the suggested usage of these features. Other usages may be possible, but will require additional user effort to create a custom simulation process.

The simulation model is implemented in the UART core's top-level HDL file; the synthesizable HDL and the simulation HDL are implemented in the same file. The simulation features are implemented using `translate on` and `translate off` synthesis directives that make certain sections of HDL code visible only to the synthesis tool.

Do not edit the simulation directives if you are using Altera's recommended simulation procedures. If you do change the simulation directives for your custom simulation flow, be aware that SOPC Builder overwrites existing files during system generation. Take precaution so that your changes are not overwritten.

For details about simulating the UART core in Nios II processor systems, refer to *AN 351: Simulating Nios II Processor Designs*.

Software Programming Model

The following sections describe the software programming model for the UART core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides hardware abstraction layer (HAL) system library drivers that enable you to access the UART core using the ANSI C standard library functions, such as `printf()` and `getchar()`.

HAL System Library Support

The Altera-provided driver implements a HAL character-mode device driver that integrates into the HAL system library for Nios II systems. HAL users should access the UART via the familiar HAL API and the ANSI C standard library, rather than accessing the UART registers. `ioctl()` requests are defined that allow HAL users to control the hardware-dependent aspects of the UART.

CAUTION

If your program uses the HAL device driver to access the UART hardware, accessing the device registers directly interferes with the correct behavior of the driver.

For Nios II processor users, the HAL system library API provides complete access to the UART core's features. Nios II programs treat the UART core as a character mode device, and send and receive data using the ANSI C standard library functions.

The driver supports the CTS/RTS control signals when they are enabled in SOPC Builder. Refer to "Driver Options: Fast Versus Small Implementations" on page 6–9.

The following code demonstrates the simplest possible usage, printing a message to `stdout` using `printf()`. In this example, the SOPC Builder system contains a UART core, and the HAL system library has been configured to use this device for `stdout`.

Example 6–1. Example: Printing Characters to a UART Core as stdout

```
#include <stdio.h>
int main ()
{
   printf("Hello world.\n");
   return 0;
}
```

The following code demonstrates reading characters from and sending messages to a UART device using the C standard library. In this example, the SOPC Builder system contains a UART core named `uart1` that is not necessarily configured as the `stdout` device. In this case, the program treats the device like any other node in the HAL file system.

Example 6–2. Example: Sending and Receiving Characters

```
/* A simple program that recognizes the characters 't' and 'v' */
#include <stdio.h>
#include <string.h>
int main ()
{
   char* msg = "Detected the character 't'.\n";
   FILE* fp;
   char prompt = 0;

   fp = fopen ("/dev/uart1", "r+"); //Open file for reading and writing
   if (fp)
   {
      while (prompt != 'v')
      {  // Loop until we receive a 'v'.
         prompt = getc(fp);  // Get a character from the UART.
         if (prompt == 't')
         {  // Print a message if character is 't'.
            fwrite (msg, strlen (msg), 1, fp);
         }
      }

      fprintf(fp, "Closing the UART file.\n");
      fclose (fp);
   }

   return 0;
}
```

For more information about the HAL system library, refer to the *Nios II Software Developer's Handbook*.

Driver Options: Fast Versus Small Implementations

To accommodate the requirements of different types of systems, the UART driver provides two variants: a fast version and a small version. The fast version is the default. Both fast and small drivers fully support the C standard library functions and the HAL API.

The fast driver is an interrupt-driven implementation, which allows the processor to perform other tasks when the device is not ready to send or receive data. Because the UART data rate is slow compared to the processor, the fast driver can provide a large performance benefit for systems that could be performing other tasks in the interim.

The small driver is a polled implementation that waits for the UART hardware before sending and receiving each character. There are two ways to enable the small footprint driver:

- Enable the small footprint setting for the HAL system library project. This option affects device drivers for all devices in the system as well.
- Specify the preprocessor option `-DALTERA_AVALON_UART_SMALL`. You can use this option if you want the small, polled implementation of the UART driver, but do not want to affect the drivers for other devices.

Refer to the help system in the Nios II IDE for details about how to set HAL properties and preprocessor options.

If the CTS/RTS flow control signals are enabled in hardware, the fast driver automatically uses them. The small driver always ignores them.

ioctl() Operations

The UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. Table 6–2 defines operation requests that the UART driver supports.

Table 6–2. UART ioctl() Operations

Request	Description
`TIOCEXCL`	Locks the device for exclusive access. Further calls to `open()` for this device will fail until either this file descriptor is closed, or the lock is released using the `TIOCNXCL` `ioctl` request. For this request to succeed there can be no other existing file descriptors for this device. The parameter `arg` is ignored.
`TIOCNXCL`	Releases a previous exclusive access lock. The parameter `arg` is ignored.

Additional operation requests are also optionally available for the fast driver only, as shown in Table 6–3. To enable these operations in your program, you must set the preprocessor option `-DALTERA_AVALON_UART_USE_IOCTL`.

Table 6–3. Optional UART ioctl() Operations for the Fast Driver Only

Request	Description
`TIOCMGET`	Returns the current configuration of the device by filling in the contents of the input termios structure. *(1)* A pointer to this structure is supplied as the value of the parameter `opt`.
`TIOCMSET`	Sets the configuration of the device according to the values contained in the input termios structure. *(1)* A pointer to this structure is supplied as the value of the parameter `arg`.

Note to Table 6–3:

(1) The termios structure is defined by the Newlib C standard library. You can find the definition in the file <*Nios II EDS install path*>**/components/altera_hal/HAL/inc/sys/termios.h**.

For details about the `ioctl()` function, refer to the *Nios II Software Developer's Handbook*.

Limitations

The HAL driver for the UART core does not support the endofpacket register. Refer to "Register Map" for details.

Software Files

The UART core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_uart_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **altera_avalon_uart.h**, **altera_avalon_uart.c**—These files implement the UART core device driver for the HAL system library.

Register Map

Programmers using the HAL API never access the UART core directly via its registers. In general, the register map is only useful to programmers writing a device driver for the core.

CAUTION

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 6–4 shows the register map for the UART core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 6–4. UART Core Register Map

Offset	Register Name	R/W	Description/Register Bits													
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	`rxdata`	RO	Reserved					*(1)*	*(1)*	Receive Data						
1	`txdata`	WO	Reserved					*(1)*	*(1)*	Transmit Data						
2	`status` *(2)*	RW	Reserved	eop	cts	dcts	(1)	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	`control`	RW	Reserved	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	`divisor` *(3)*	RW	Baud Rate Divisor													
5	`endof-packet` *(3)*	RW	Reserved					*(1)*	*(1)*	End-of-Packet Value						

Notes to Table 6–4:

(1) These bits may or may not exist, depending on the **Data Width** hardware option. If they do not exist, they read zero, and writing has no effect.

(2) Writing zero to the `status` register clears the `dcts`, `e`, `toe`, `roe`, `brk`, `fe`, and `pe` bits.

(3) This register may or may not exist, depending on hardware configuration options. If it does not exist, reading returns an undefined value and writing has no effect.

Some registers and bits are optional. These registers and bits exists in hardware only if it was enabled at system generation time. Optional registers and bits are noted in the following sections.

rxdata Register

The `rxdata` register holds data received via the `RXD` input. When a new character is fully received via the `RXD` input, it is transferred into the `rxdata` register, and the `status` register's `rrdy` bit is set to 1. The `status` register's `rrdy` bit is set to 0 when the `rxdata` register is read. If a character is transferred into the `rxdata` register while the `rrdy` bit is already set (in other words, the previous character was not retrieved), a receiver-overrun error occurs and the `status` register's `roe` bit is set to 1. New characters are always transferred into the `rxdata` register, regardless of whether the previous character was read. Writing data to the `rxdata` register has no effect.

txdata Register

Avalon-MM master peripherals write characters to be transmitted into the `txdata` register. Characters should not be written to `txdata` until the transmitter is ready for a new character, as indicated by the `TRDY` bit in the `status` register. The `TRDY` bit is set to 0 when a character is written into the `txdata` register. The `TRDY` bit is set to 1 when the character is transferred from the `txdata` register into the transmitter shift register. If a character is written to the `txdata` register when `TRDY` is 0, the result is undefined. Reading the `txdata` register returns an undefined value.

For example, assume the transmitter logic is idle and an Avalon-MM master peripheral writes a first character into the `txdata` register. The `TRDY` bit is set to 0, then set to 1 when the character is transferred into the transmitter shift register. The master can then write a second character into the `txdata` register, and the `TRDY` bit is set to 0 again. However, this time the shift register is still busy shifting out the first character to the `TXD` output. The `TRDY` bit is not set to 1 until the first character is fully shifted out and the second character is automatically transferred into the transmitter shift register.

status Register

The `status` register consists of individual bits that indicate particular conditions inside the UART core. Each status bit is associated with a corresponding interrupt-enable bit in the `control` register. The `status` register can be read at any time. Reading does not change the value of any of the bits. Writing zero to the `status` register clears the `DCTS`, `E`, `TOE`, `ROE`, `BRK`, `FE`, and `PE` bits.

The `status` register bits are shown in Table 6–5.

Table 6–5. status Register Bits (Part 1 of 2)

Bit	Name	Access	Description
0 *(1)*	`PE`	RC	Parity error. A parity error occurs when the received parity bit has an unexpected (incorrect) logic level. The `PE` bit is set to 1 when the core receives a character with an incorrect parity bit. The `PE` bit stays set to 1 until it is explicitly cleared by a write to the `status` register. When the `PE` bit is set, reading from the `rxdata` register produces an undefined value. If the **Parity** hardware option is not enabled, no parity checking is performed and the `PE` bit always reads 0. Refer to "Data Bits, Stop Bits, Parity" on page 6–5.
1	`FE`	RC	Framing error. A framing error occurs when the receiver fails to detect a correct stop bit. The `FE` bit is set to 1 when the core receives a character with an incorrect stop bit. The `FE` bit stays set to 1 until it is explicitly cleared by a write to the `status` register. When the `FE` bit is set, reading from the `rxdata` register produces an undefined value.
2	`BRK`	RC	Break detect. The receiver logic detects a break when the `RXD` pin is held low (logic 0) continuously for longer than a full-character time (data bits, plus start, stop, and parity bits). When a break is detected, the `BRK` bit is set to 1. The `BRK` bit stays set to 1 until it is explicitly cleared by a write to the `status` register.
3	`ROE`	RC	Receive overrun error. A receive-overrun error occurs when a newly received character is transferred into the `rxdata` holding register before the previous character is read (in other words, while the `RRDY` bit is 1). In this case, the `ROE` bit is set to 1, and the previous contents of `rxdata` are overwritten with the new character. The `ROE` bit stays set to 1 until it is explicitly cleared by a write to the `status` register.

Table 6–5. status Register Bits (Part 2 of 2)

Bit	Name	Access	Description
4	TOE	RC	Transmit overrun error. A transmit-overrun error occurs when a new character is written to the txdata holding register before the previous character is transferred into the shift register (in other words, while the TRDY bit is 0). In this case the TOE bit is set to 1. The TOE bit stays set to 1 until it is explicitly cleared by a write to the status register.
5	TMT	R	Transmit empty. The TMT bit indicates the transmitter shift register's current state. When the shift register is in the process of shifting a character out the TXD pin, TMT is set to 0. When the shift register is idle (in other words, a character is not being transmitted) the TMT bit is 1. An Avalon-MM master peripheral can determine if a transmission is completed (and received at the other end of a serial link) by checking the TMT bit.
6	TRDY	R	Transmit ready. The TRDY bit indicates the txdata holding register's current state. When the txdata register is empty, it is ready for a new character, and TRDY is 1. When the txdata register is full, TRDY is 0. An Avalon-MM master peripheral must wait for TRDY to be 1 before writing new data to txdata.
7	RRDY	R	Receive character ready. The RRDY bit indicates the rxdata holding register's current state. When the rxdata register is empty, it is not ready to be read and RRDY is 0. When a newly received value is transferred into the rxdata register, RRDY is set to 1. Reading the rxdata register clears the RRDY bit to 0. An Avalon-MM master peripheral must wait for RRDY to equal 1 before reading the rxdata register.
8	E	RC	Exception. The E bit indicates that an exception condition occurred. The E bit is a logical-OR of the TOE, ROE, BRK, FE, and PE bits. The E bit and its corresponding interrupt-enable bit (IE) bit in the control register provide a convenient method to enable/disable IRQs for all error conditions. The E bit is set to 0 by a write operation to the status register.
10 *(1)*	DCTS	RC	Change in clear to send (CTS) signal. The DCTS bit is set to 1 whenever a logic-level transition is detected on the CTS_N input port (sampled synchronously to the Avalon-MM clock). This bit is set by both falling and rising transitions on CTS_N. The DCTS bit stays set to 1 until it is explicitly cleared by a write to the status register. If the **Flow Control** hardware option is not enabled, the DCTS bit always reads 0. Refer to "Flow Control" on page 6–5.
11 *(1)*	CTS	R	Clear-to-send (CTS) signal. The CTS bit reflects the CTS_N input's instantaneous state (sampled synchronously to the Avalon-MM clock). The CTS_N input has no effect on the transmit or receive processes. The only visible effect of the CTS_N input is the state of the CTS and DCTS bits, and an IRQ that can be generated when the control register's idcts bit is enabled. If the **Flow Control** hardware option is not enabled, the CTS bit always reads 0. Refer to "Flow Control" on page 6–5.
12 *(1)*	EOP	R	End of packet encountered. The EOP bit is set to 1 by one of the following events: ■ An EOP character is written to txdata ■ An EOP character is read from rxdata The EOP character is determined by the contents of the endofpacket register. The EOP bit stays set to 1 until it is explicitly cleared by a write to the status register. If the **Include End-of-Packet Register** hardware option is not enabled, the EOP bit always reads 0. Refer to "Streaming Data (DMA) Control" on page 6–6.

Note to Table 6–5:

(1) This bit is optional and may not exist in hardware.

control Register

The `control` register consists of individual bits, each controlling an aspect of the UART core's operation. The value in the `control` register can be read at any time.

Each bit in the `control` register enables an IRQ for a corresponding bit in the `status` register. When both a status bit and its corresponding interrupt-enable bit are 1, the core generates an IRQ.

The control register bits are shown in Table 6–6.

Table 6–6. control Register Bits

Bit	Name	Access	Description
0	IPE	RW	Enable interrupt for a parity error.
1	IFE	RW	Enable interrupt for a framing error.
2	IBRK	RW	Enable interrupt for a break detect.
3	IROE	RW	Enable interrupt for a receiver overrun error.
4	ITOE	RW	Enable interrupt for a transmitter overrun error.
5	ITMT	RW	Enable interrupt for a transmitter shift register empty.
6	ITRDY	RW	Enable interrupt for a transmission ready.
7	IRRDY	RW	Enable interrupt for a read ready.
8	IE	RW	Enable interrupt for an exception.
9	TRBK	RW	Transmit break. The TRBK bit allows an Avalon-MM master peripheral to transmit a break character over the TXD output. The TXD signal is forced to 0 when the TRBK bit is set to 1. The TRBK bit overrides any logic level that the transmitter logic would otherwise drive on the TXD output. The TRBK bit interferes with any transmission in process. The Avalon-MM master peripheral must set the TRBK bit back to 0 after an appropriate break period elapses.
10	IDCTS	RW	Enable interrupt for a change in CTS signal.
11 *(1)*	RTS	RW	Request to send (RTS) signal. The RTS bit directly feeds the RTS_N output. An Avalon-MM master peripheral can write the RTS bit at any time. The value of the RTS bit only affects the RTS_N output; it has no effect on the transmitter or receiver logic. Because the RTS_N output is logic negative, when the RTS bit is 1, a low logic-level (0) is driven on the RTS_N output. If the **Flow Control** hardware option is not enabled, the RTS bit always reads 0, and writing has no effect. Refer to "Flow Control" on page 6–5.
12	IEOP	RW	Enable interrupt for end-of-packet condition.

Note to Table 6–6:

(1) This bit is optional and may not exist in hardware.

divisor Register (Optional)

The value in the `divisor` register is used to generate the baud rate clock. The effective baud rate is determined by the formula:

Baud Rate = (Clock frequency) / (divisor + 1)

The `divisor` register is an optional hardware feature. If the **Baud Rate Can Be Changed By Software** hardware option is not enabled, the `divisor` register does not exist. In this case, writing `divisor` has no effect, and reading `divisor` returns an undefined value. For more information, refer to "Baud Rate Options" on page 6–4.

endofpacket Register (Optional)

The value in the `endofpacket` register determines the end-of-packet character for variable-length DMA transactions. After reset, the default value is zero, which is the ASCII null character (\0). For more information, refer to Table 6–5 on page 6–12 for the description for the `EOP` bit.

The `endofpacket` register is an optional hardware feature. If the **Include end-of-packet register** hardware option is not enabled, the `endofpacket` register does not exist. In this case, writing `endofpacket` has no effect, and reading returns an undefined value.

Interrupt Behavior

The UART core outputs a single IRQ signal to the Avalon-MM interface, which can connect to any master peripheral in the system, such as a Nios II processor. The master peripheral must read the `status` register to determine the cause of the interrupt.

Every interrupt condition has an associated bit in the `status` register and an interrupt-enable bit in the `control` register. When any of the interrupt conditions occur, the associated `status` bit is set to 1 and remains set until it is explicitly acknowledged. The IRQ output is asserted when any of the status bits are set while the corresponding interrupt-enable bit is 1. A master peripheral can acknowledge the IRQ by clearing the `status` register.

At reset, all interrupt-enable bits are set to 0; therefore, the core cannot assert an IRQ until a master peripheral sets one or more of the interrupt-enable bits to 1.

All possible interrupt conditions are listed with their associated status and control (interrupt-enable) bits in Table 6–5 on page 6–16 and Table 6–6 on page 6–18. Details of each interrupt condition are provided in the `status` bit descriptions.

Referenced Documents

This chapter references the following documents:

- *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*
- *AN 351: Simulating Nios II Embedded Processor Designs*
- *Avalon Interface Specifications*
- *Nios II Software Developer's Handbook*
- "Interval Timer Core" on page 27–1
- *AN 42: Metastability in Altera Devices*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 6–7 shows the revision history for this chapter.

Table 6–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	Added description of a new parameter, **Synchronizer stages**.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

7. SPI Core

Core Overview

SPI is an industry-standard serial protocol commonly used in embedded systems to connect microprocessors to a variety of off-chip sensor, conversion, memory, and control devices. The SPI core with Avalon® interface implements the SPI protocol and provides an Avalon Memory-Mapped (Avalon-MM) interface on the back end.

The SPI core can implement either the master or slave protocol. When configured as a master, the SPI core can control up to 32 independent SPI slaves. The width of the receive and transmit registers are configurable between 1 and 32 bits. Longer transfer lengths can be supported with software routines. The SPI core provides an interrupt output that can flag an interrupt whenever a transfer completes.

The SPI core is SOPC Builder ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Instantiating the SPI Core in SOPC Builder" on page 7–5
- "Device Support" on page 7–8
- "Software Programming Model" on page 7–8

Functional Description

The SPI core communicates using two data lines, a control line, and a synchronization clock:

- Master Out Slave In (`mosi`)—Output data from the master to the inputs of the slaves
- Master In Slave Out (`miso`)—Output data from a slave to the input of the master
- Serial Clock (`sclk`)—Clock driven by the master to slaves, used to synchronize the data bits
- Slave Select (`ss_n`)— Select signal (active low) driven by the master to individual slaves, used to select the target slave

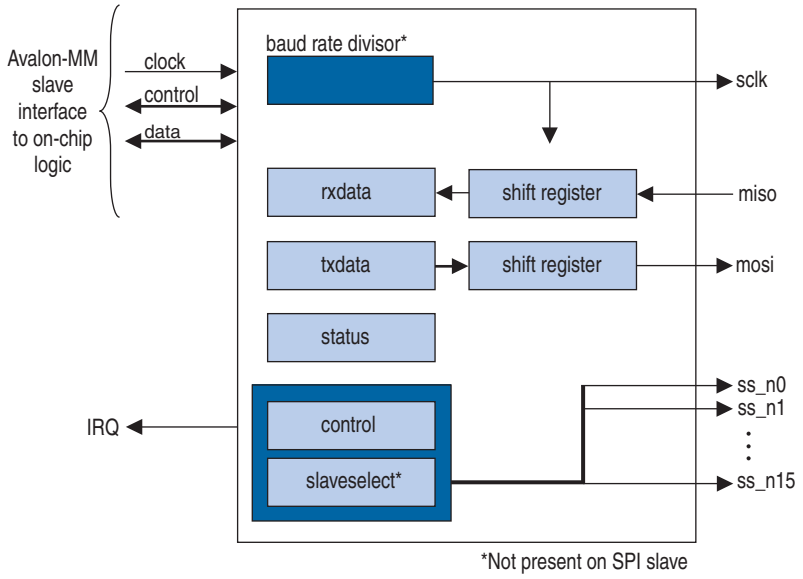
The SPI core has the following user-visible features:

- A memory-mapped register space comprised of five registers: `rxdata`, `txdata`, `status`, `control`, and `slaveselect`
- Four SPI interface ports: `sclk`, `ss_n`, `mosi`, and `miso`

The registers provide an interface to the SPI core and are visible via the Avalon-MM slave port. The `sclk`, `ss_n`, `mosi`, and `miso` ports provide the hardware interface to other SPI devices. The behavior of `sclk`, `ss_n`, `mosi`, and `miso` depends on whether the SPI core is configured as a master or slave.

Figure 7–1 shows a block diagram of the SPI core in master mode.

Figure 7–1. SPI Core Block Diagram (Master Mode)



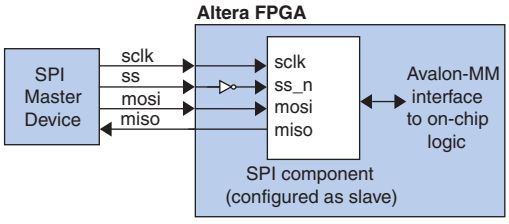
The SPI core logic is synchronous to the clock input provided by the Avalon-MM interface. When configured as a master, the core divides the Avalon-MM clock to generate the SCLK output. When configured as a slave, the core's receive logic is synchronized to SCLK input. The core's Avalon-MM interface is capable of Avalon-MM transfers with flow control. The SPI core can be used in conjunction with a DMA controller with flow control to automate continuous data transfers between, for example, the SPI core and memory.

For more details, refer to "Interval Timer Core" on page 27–1.

Example Configurations

Figure 7–1 and Figure 7–2 show two possible configurations. In Figure 7–2, the SPI core provides a slave interface to an off-chip SPI master.

Figure 7–2. SPI Core Configured as a Slave



In Figure 7–1, the SPI core provides a master interface driving multiple off-chip slave devices. Each slave device in Figure 7–1 must tristate its `miso` output whenever its select signal is not asserted.

The `ss_n` signal is active-low. However, any signal can be inverted inside the FPGA, allowing the slave-select signals to be either active high or active low.

Transmitter Logic

The SPI core transmitter logic consists of a transmit holding register (`txdata`) and transmit shift register, each *n* bits wide. The register width *n* is specified at system generation time, and can be any integer value from 8 to 32. After a master peripheral writes a value to the `txdata` register, the value is copied to the shift register and then transmitted when the next operation starts.

The shift register and the `txdata` register provide double buffering during data transmission. A new value can be written into the `txdata` register while the previous data is being shifted out of the shift register. The transmitter logic automatically transfers the `txdata` register to the shift register whenever a serial shift operation is not currently in process.

In master mode, the transmit shift register directly feeds the `mosi` output. In slave mode, the transmit shift register directly feeds the `miso` output. Data shifts out LSB first or MSB first, depending on the configuration of the SPI core.

Receiver Logic

The SPI core receive logic consists of a receive holding register (`rxdata`) and receive shift register, each *n* bits wide. The register width *n* is specified at system generation time, and can be any integer value from 8 to 32. A master peripheral reads received data from the `rxdata` register after the shift register has captured a full *n*-bit value of data.

The shift register and the `rxdata` register provide double buffering while receiving data. The `rxdata` register can hold a previously received data value while subsequent new data is shifting into the shift register. The receiver logic automatically transfers the shift register content to the `rxdata` register when a serial shift operation completes.

In master mode, the shift register is fed directly by the `miso` input. In slave mode, the shift register is fed directly by the `mosi` input. The receiver logic expects input data to arrive LSB first or MSB first, depending on the configuration of the SPI core.

Master and Slave Modes

At system generation time, the designer configures the SPI core in either master mode or slave mode. The mode cannot be switched at runtime.

Master Mode Operation

In master mode, the SPI ports behave as shown in Table 7–1.

Table 7–1. Master Mode Port Configurations (Part 1 of 2)

Name	Direction	Description
`mosi`	output	Data output to slave(s)
`miso`	input	Data input from slave(s)

Table 7–1. Master Mode Port Configurations (Part 2 of 2)

Name	Direction	Description
sclk	output	Synchronization clock to all slaves
ss_n*M*	output	Slave select signal to slave *M*, where M is a number between 0 and 31.

In master mode, an intelligent host (for example, a microprocessor) configures the SPI core using the control and slaveselect registers, and then writes data to the txdata buffer to initiate a transaction. A master peripheral can monitor the status of the transaction by reading the status register. A master peripheral can enable interrupts to notify the host whenever new data is received (for example, a transfer has completed), or whenever the transmit buffer is ready for new data.

The SPI protocol is full duplex, so every transaction both sends and receives data at the same time. The master transmits a new data bit on the mosi output and the slave drives a new data bit on the miso input for each active edge of sclk. The SPI core divides the Avalon-MM system clock using a clock divider to generate the sclk signal.

When the SPI core is configured to interface with multiple slaves, the core has one ss_n signal for each slave. During a transfer, the master asserts ss_n to each slave specified in the slaveselect register. Note that there can be no more than one slave transmitting data during any particular transfer, or else there will be a contention on the miso input. The number of slave devices is specified at system generation time.

Slave Mode Operation

In slave mode, the SPI ports behave as shown in Table 7–2.

Table 7–2. Slave Mode Port Configurations

Name	Direction	Description
mosi	input	Data input from the master
miso	output	Data output to the master
sclk	input	Synchronization clock
ss_n	input	Select signal

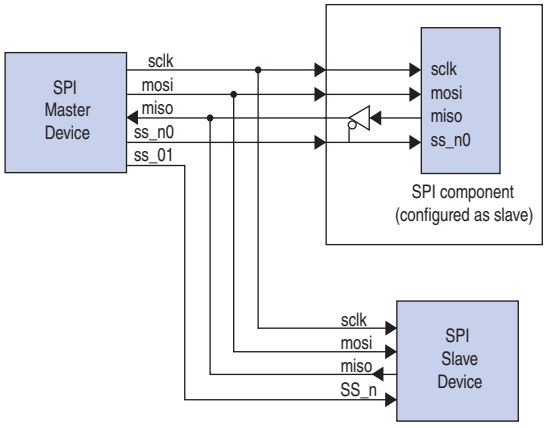
In slave mode, the SPI core simply waits for the master to initiate transactions. Before a transaction begins, the slave logic continuously polls the ss_n input. When the master asserts ss_n, the slave logic immediately begins sending the transmit shift register contents to the miso output. The slave logic also captures data on the mosi input, and fills the receive shift register simultaneously. After a word is received by the slave, the master must de-assert the ss_n signal and reasserts the signal again when the next word is ready to be sent.

An intelligent host such as a microprocessor writes data to the txdata registers, so that it is transmitted the next time the master initiates an operation. A master peripheral reads received data from the rxdata register. A master peripheral can enable interrupts to notify the host whenever new data is received, or whenever the transmit buffer is ready for new data.

Multi-Slave Environments

When `ss_n` is not asserted, typical SPI cores set their `miso` output pins to high impedance. The Altera®-provided SPI slave core drives an undefined high or low value on its `miso` output when not selected. Special consideration is necessary to avoid signal contention on the `miso` output, if the SPI core in slave mode is connected to an off-chip SPI master device with multiple slaves. In this case, the `ss_n` input should be used to control a tristate buffer on the `miso` signal. Figure 7–3 shows an example of the SPI core in slave mode in an environment with two slaves.

Figure 7–3. SPI Core in a Multi-Slave Environment



Avalon-MM Interface

The SPI core's Avalon-MM interface consists of a single Avalon-MM slave port. In addition to fundamental slave read and write transfers, the SPI core supports Avalon-MM read and write transfers with flow control. The flow control is disabled when:

- the option to disable flow control is turned on, or
- the option to disable flow control is turned off and the master does not support flow control.

Instantiating the SPI Core in SOPC Builder

You can use the MegaWizard™ interface for the SPI core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.

Master/Slave Settings

The designer can select either master mode or slave mode to determine the role of the SPI core. When master mode is selected, the following options are available: **Number of select (SS_n) signals**, **SPI clock rate**, and **Specify delay**.

Number of Select (SS_n) Signals

This setting specifies the number of slaves the SPI master connects to. The range is 1 to 32. The SPI master core presents a unique `ss_n` signal for each slave.

SPI Clock (sclk) Rate

This setting determines the rate of the `sclk` signal that synchronizes data between master and slaves. The target clock rate can be specified in units of Hz, kHz or MHz. The SPI master core uses the Avalon-MM system clock and a clock divisor to generate `sclk`.

The actual frequency of `sclk` may not exactly match the desired target clock rate. The achievable clock values are:

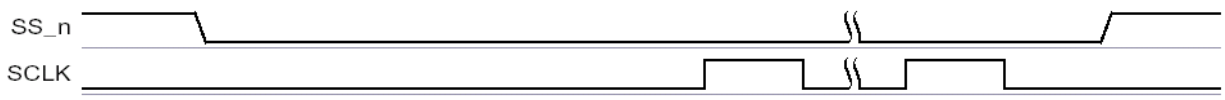
<Avalon-MM system clock frequency> / [2, 4, 6, 8, ...]

The actual frequency achieved will not be greater than the specified target value.

Specify Delay

Turning on this option causes the SPI master to add a time delay between asserting the `ss_n` signal and shifting the first bit of data. This delay is required by certain SPI slave devices. If the delay option is on, you must also specify the delay time in units of ns, µs or ms. An example is shown in Figure 7–4.

Figure 7–4. Time Delay Between Asserting ss_n and Toggling sclk



The delay generation logic uses a granularity of half the period of `sclk`. The actual delay achieved is the desired target delay rounded up to the nearest multiple of half the `sclk` period, as shown in Equation 7–1 and Equation 7–2:

Equation 7–1.

$$p = \frac{1}{2} \times (\text{period of sclk})$$

Equation 7–2.

$$\text{actual delay} = \text{ceiling} \times \left(\frac{\text{desired delay}}{p}\right) \times p$$

Data Register Settings

The data register settings affect the size and behavior of the data registers in the SPI core. There are two data register settings:

- **Width**—This setting specifies the width of `rxdata`, `txdata`, and the receive and transmit shift registers. The range is from 1 to 32.
- **Shift direction**—This setting determines the direction that data shifts (MSB first or LSB first) into and out of the shift registers.

Timing Settings

The timing settings affect the timing relationship between the `ss_n`, `sclk`, `mosi` and `miso` signals. In this discussion the `mosi` and `miso` signals are referred to generically as *data*. There are two timing settings:

- **Clock polarity**—This setting can be 0 or 1. When clock polarity is set to 0, the idle state for `sclk` is low. When clock polarity is set to 1, the idle state for `sclk` is high.
- **Clock phase**—This setting can be 0 or 1. When clock phase is 0, data is latched on the leading edge of `sclk`, and data changes on trailing edge. When clock phase is 1, data is latched on the trailing edge of `sclk`, and data changes on the leading edge.

Figure 7–5 through Figure 7–8 demonstrate the behavior of signals in all possible cases of clock polarity and clock phase.

Figure 7–5. Clock Polarity = 0, Clock Phase = 0

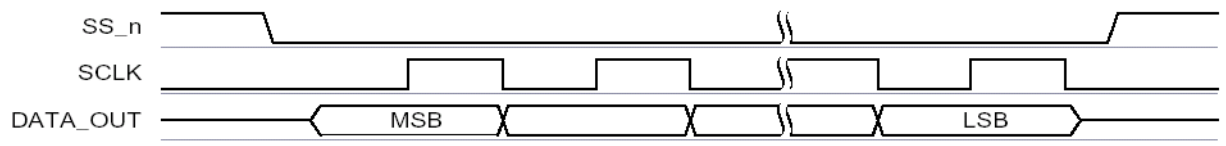


Figure 7–6. Clock Polarity = 0, Clock Phase = 1

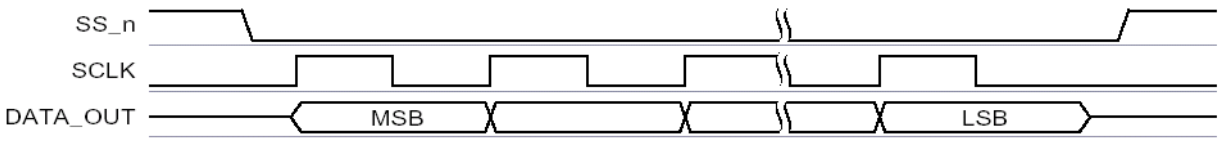


Figure 7–7. Clock Polarity = 1, Clock Phase = 0

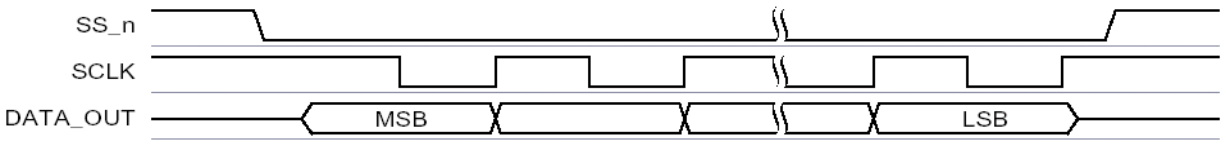
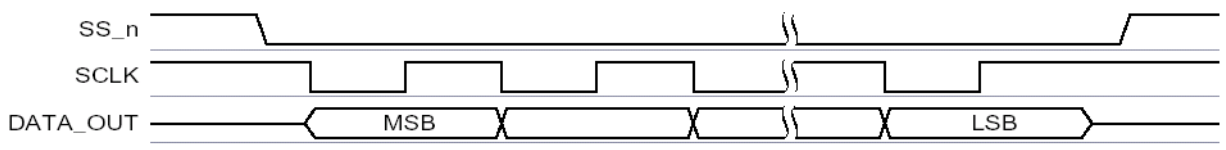


Figure 7–8. Clock Polarity = 1, Clock Phase = 1



Device Support

The SPI core supports all Altera® device families.

Software Programming Model

The following sections describe the software programming model for the SPI core, including the register map and software constructs used to access the hardware. For Nios® II processor users, Altera provides the HAL system library header file that defines the SPI core registers. The SPI core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library. Altera provides a routine to access the SPI hardware that is specific to the SPI core.

Hardware Access Routines

Altera provides one access routine, `alt_avalon_spi_command()`, that provides general-purpose access to the SPI core that is configured as a master.

alt_avalon_spi_command()

Prototype:

```
int alt_avalon_spi_command(alt_u32 base, alt_u32 slave,
                           alt_u32 write_length,
                           const alt_u8* wdata,
                           alt_u32 read_length,
                           alt_u8* read_data,
                           alt_u32 flags)
```

Thread-safe: No.

Available from ISR: No.

Include: <altera_avalon_spi.h>

Description: This function performs a control sequence on the SPI bus. It supports only SPI masters with data width less than or equal to 8 bits. A single call to this function writes a data buffer of arbitrary length to the `mosi` port, and then reads back an arbitrary amount of data from the `miso` port. The function performs the following actions:

(1) Asserts the slave select output for the specified slave. The first slave select output is 0.

(2) Transmits `write_length` bytes of data from `wdata` through the SPI interface, discarding the incoming data on the `miso` port.

(3) Reads `read_length` bytes of data and stores the data into the buffer pointed to by `read_data`. The `mosi` port is set to zero during the read transaction.

(4) De-asserts the slave select output, unless the flags field contains the value `ALT_AVALON_SPI_COMMAND_MERGE`. If you want to transmit from scattered buffers, call the function multiple times and specify the merge flag on all the accesses except the last.

To access the SPI bus from more than one thread, you must use a semaphore or mutex to ensure that only one thread is executing within this function at any time.

Returns: The number of bytes stored in the `read_data` buffer.

Software Files

The SPI core is accompanied by the following software files. These files provide a low-level interface to the hardware.

- **altera_avalon_spi.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_spi.c**—This file implements low-level routines to access the hardware.

Register Map

An Avalon-MM master peripheral controls and communicates with the SPI core via the six 32-bit registers, shown in Table 7–3. The table assumes an *n*-bit data width for `rxdata` and `txdata`.

Table 7–3. Register Map for SPI Master Device

Internal Address	Register Name	Type [R/W]	32..11	10	9	8	7	6	5	4	3	2	1	0
0	`rxdata` *(1)*	R	`RXDATA` (n-1..0)											
1	`txdata` *(1)*	W	`TXDATA` (n-1..0)											

Table 7–3. Register Map for SPI Master Device

Internal Address	Register Name	Type [R/W]	32..11	10	9	8	7	6	5	4	3	2	1	0
2	`status` *(2)*	R/W				E	RRDY	TRDY	TMT	TOE	ROE			
3	`control`	R/W		SSO *(3)*		IE	IRRDY	ITRDY		ITOE	IROE			
4	Reserved	—												
5	`slaveselect` *(3)*	R/W	Slave Select Mask											

Notes to Table 7–3:

(1) Bits 31 to *n* are undefined when *n* is less than 32.

(2) A write operation to the `status` register clears the `ROE`, `TOE`, and `E` bits.

(3) Present only in master mode.

Reading undefined bits returns an undefined value. Writing to undefined bits has no effect.

rxdata Register

A master peripheral reads received data from the `rxdata` register. When the receive shift register receives a full *n* bits of data, the `status` register's `RRDY` bit is set to 1 and the data is transferred into the `rxdata` register. Reading the `rxdata` register clears the `RRDY` bit. Writing to the `rxdata` register has no effect.

New data is always transferred into the `rxdata` register, whether or not the previous data was retrieved. If `RRDY` is 1 when data is transferred into the `rxdata` register (that is, the previous data was not retrieved), a receive-overrun error occurs and the `status` register's `ROE` bit is set to 1. In this case, the contents of `rxdata` are undefined.

txdata Register

A master peripheral writes data to be transmitted into the `txdata` register. When the `status` register's `TRDY` bit is 1, it indicates that the `txdata` register is ready for new data. The `TRDY` bit is set to 0 whenever the `txdata` register is written. The `TRDY` bit is set to 1 after data is transferred from the `txdata` register into the transmitter shift register, which readies the `txdata` holding register to receive new data.

A master peripheral should not write to the `txdata` register until the transmitter is ready for new data. If `TRDY` is 0 and a master peripheral writes new data to the `txdata` register, a transmit-overrun error occurs and the `status` register's `TOE` bit is set to 1. In this case, the new data is ignored, and the content of `txdata` remains unchanged.

As an example, assume that the SPI core is idle (that is, the `txdata` register and transmit shift register are empty), when a CPU writes a data value into the `txdata` holding register. The `TRDY` bit is set to 0 momentarily, but after the data in `txdata` is transferred into the transmitter shift register, `TRDY` returns to 1. The CPU writes a second data value into the `txdata` register, and again the `TRDY` bit is set to 0. This time the shift register is still busy transferring the original data value, so the `TRDY` bit remains at 0 until the shift operation completes. When the operation completes, the second data value is transferred into the transmitter shift register and the `TRDY` bit is again set to 1.

status Register

The `status` register consists of bits that indicate status conditions in the SPI core. Each bit is associated with a corresponding interrupt-enable bit in the `control` register, as discussed in "control Register" on page 7–12. A master peripheral can read `status` at any time without changing the value of any bits. Writing `status` does clear the `ROE`, `TOE` and `E` bits. Table 7–4 describes the individual bits of the `status` register.

Table 7–4. status Register Bits

#	Name	Description
3	`ROE`	**Receive-overrun error** The `ROE` bit is set to 1 if new data is received while the `rxdata` register is full (that is, while the `RRDY` bit is 1). In this case, the new data overwrites the old. Writing to the `status` register clears the `ROE` bit to 0.
4	`TOE`	**Transmitter-overrun error** The `TOE` bit is set to 1 if new data is written to the `txdata` register while it is still full (that is, while the `TRDY` bit is 0). In this case, the new data is ignored. Writing to the `status` register clears the `TOE` bit to 0.
5	`TMT`	**Transmitter shift-register empty** In master mode, the `TMT` bit is set to 0 when a transaction is in progress and set to 1 when the shift register is empty. In slave mode, the `TMT` bit is set to 0 when the slave is selected (`SS_n` is low) or when the SPI Slave register interface is not ready to receive data.
6	`TRDY`	**Transmitter ready** The `TRDY` bit is set to 1 when the `txdata` register is empty.
7	`RRDY`	**Receiver ready** The `RRDY` bit is set to 1 when the `rxdata` register is full.
8	`E`	**Error** The `E` bit is the logical OR of the `TOE` and `ROE` bits. This is a convenience for the programmer to detect error conditions. Writing to the `status` register clears the `E` bit to 0.

control Register

The control register consists of data bits to control the SPI core's operation. A master peripheral can read control at any time without changing the value of any bits.

Most bits (IROE, ITOE, ITRDY, IRRDY, and IE) in the control register control interrupts for status conditions represented in the status register. For example, bit 1 of status is ROE (receiver-overrun error), and bit 1 of control is IROE, which enables interrupts for the ROE condition. The SPI core asserts an interrupt request when the corresponding bits in status and control are both 1.

The control register bits are shown in Table 7–5.

Table 7–5. control Register Bits

#	Name	Description
3	IROE	Setting IROE to 1 enables interrupts for receive-overrun errors.
4	ITOE	Setting ITOE to 1 enables interrupts for transmitter-overrun errors.
6	ITRDY	Setting ITRDY to 1 enables interrupts for the transmitter ready condition.
7	IRRDY	Setting IRRDY to 1 enables interrupts for the receiver ready condition.
8	IE	Setting IE to 1 enables interrupts for any error condition.
10	SSO	Setting SSO to 1 forces the SPI core to drive its ss_n outputs, regardless of whether a serial shift operation is in progress or not. The slaveselect register controls which ss_n outputs are asserted. SSO can be used to transmit or receive data of arbitrary size, for example, greater than 32 bits.

After reset, all bits of the control register are set to 0. All interrupts are disabled and no ss_n signals are asserted.

slaveselect Register

The slaveselect register is a bit mask for the ss_n signals driven by an SPI master. During a serial shift operation, the SPI master selects only the slave device(s) specified in the slaveselect register.

The slaveselect register is only present when the SPI core is configured in master mode. There is one bit in slaveselect for each ss_n output, as specified by the designer at system generation time.

A master peripheral can set multiple bits of slaveselect simultaneously, causing the SPI master to simultaneously select multiple slave devices as it performs a transaction. For example, to enable communication with slave devices 1, 5, and 6, set bits 1, 5, and 6 of slaveselect. However, consideration is necessary to avoid signal contention between multiple slaves on their miso outputs.

Upon reset, bit 0 is set to 1, and all other bits are cleared to 0. Thus, after a device reset, slave device 0 is automatically selected.

Referenced Documents

This chapter references the following documents:

- *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*
- *"Interval Timer Core" on page 27–1*

Document Revision History

Table 7–6 shows the revision history for this chapter.

Table 7–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	■ Revised register width in transmitter logic and receiver logic. ■ Added description on the disable flow control option. ■ Added R/W column in Table 7–3.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Updated the width of the parameters and signals from 16 to 32.	—
May 2008 v8.0.0	Updated the description of the `TMT` bit.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

8. Optrex 16207 LCD Controller Core

Core Overview

The Optrex 16207 LCD controller core with Avalon® Interface (LCD controller core) provides the hardware interface and software driver required for a Nios® II processor to display characters on an Optrex 16207 (or equivalent) 16×2-character LCD panel. Device drivers are provided in the HAL system library for the Nios II processor. Nios II programs access the LCD controller as a character mode device using ANSI C standard library routines, such as `printf()`. The LCD controller is SOPC Builder-ready, and integrates easily into any SOPC Builder-generated system.

The Nios II Embedded Design Suite (EDS) includes an Optrex LCD module and provide several ready-made example designs that display text on the Optrex 16207 via the LCD controller. For details about the Optrex 16207 LCD module, see the manufacturer's *Dot Matrix Character LCD Module User's Manual* available at www.optrex.com.

This chapter contains the following sections:

- "Functional Description"
- "Device and Tools Support" on page 8–2
- "Instantiating the Core in SOPC Builder" on page 8–2
- "Software Programming Model" on page 8–2

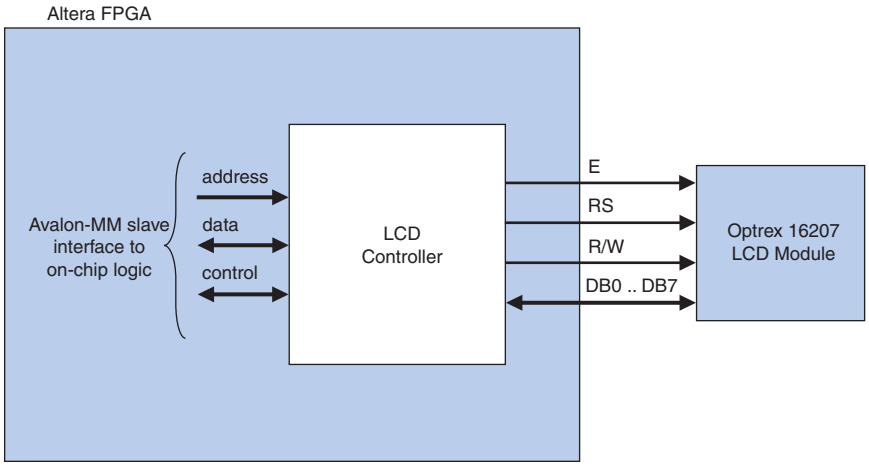
Functional Description

The LCD controller core consists of two user-visible components:

- Eleven signals that connect to pins on the Optrex 16207 LCD panel—These signals are defined in the Optrex 16207 data sheet.
  - `E`—Enable (output)
  - `RS`—Register Select (output)
  - `R/W`—Read or Write (output)
  - `DB0` through `DB7`—Data Bus (bidirectional)
- An Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to 4 registers.

Figure 8–1 shows a block diagram of the LCD controller core.

Figure 8–1. LCD Controller Block Diagram



Device and Tools Support

The LCD controller core supports all Altera device families. The LCD controller drivers support the Nios II processor.

Instantiating the Core in SOPC Builder

You can add the LCD controller core from the **System Contents** tab in SOPC Builder. In SOPC Builder, the LCD controller core has the name Character LCD (16×2, Optrex 16207). There are no user-configurable settings for this component.

Software Programming Model

This section describes the software programming model for the LCD controller.

HAL System Library Support

Altera provides HAL system library drivers for the Nios II processor that enable you to access the LCD controller using the ANSI C standard library functions. The Altera-provided drivers integrate into the HAL system library for Nios II systems. The LCD driver is a standard character-mode device, as described in the *Nios II Software Developer's Handbook*. Therefore, using `printf()` is the easiest way to write characters to the display.

The LCD driver requires that the HAL system library include the system clock driver.

Displaying Characters on the LCD

The driver implements VT100 terminal-like behavior on a miniature scale for the 16×2 screen. Characters written to the LCD controller are stored to an 80-column × 2-row buffer maintained by the driver. As characters are written, the cursor position is updated. Visible characters move the cursor position to the right. Any visible characters written to the right of the buffer are discarded. The line feed character (`\n`) moves the cursor down one line and to the left-most column.

The buffer is scrolled up as soon as a printable character is written onto the line below the bottom of the buffer. Rows do not scroll as soon as the cursor moves down to allow the maximum useful information in the buffer to be displayed.

If the visible characters in the buffer fit on the display, all characters are displayed. If the buffer is wider than the display, the display scrolls horizontally to display all the characters. Different lines scroll at different speeds, depending on the number of characters in each line of the buffer.

The LCD driver supports a small subset of ANSI and VT100 escape sequences that can be used to control the cursor position, and clear the display as shown in Table 8–1.

Table 8–1. Escape Sequence Supported by the LCD Controller

Sequence	Meaning
`BS (\b)`	Moves the cursor to the left by one character.
`CR (\r)`	Moves the cursor to the start of the current line.
`LF (\n)`	Moves the cursor to the start of the line and move it down one line.
`ESC( (\x1B)`	Starts a VT100 control sequence.
`ESC [ <y> ; <x> H`	Moves the cursor to the y, x position specified – positions are counted from the top left which is 1;1.
`ESC [ K`	Clears from current cursor position to end of line.
`ESC [ 2 J`	Clears the whole screen.

The LCD controller is an output-only device. Therefore, attempts to read from it returns immediately indicating that no characters have been received.

The LCD controller drivers are not included in the system library when the **Reduced device drivers** option is enabled for the system library. If you want to use the LCD controller while using small drivers for other devices, add the preprocessor option—`DALT_USE_LCD_16207` to the preprocessor options.

Software Files

The LCD controller is accompanied by the following software files. These files define the low-level interface to the hardware and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_lcd_16207_regs.h** — This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_lcd_16207.h**, **altera_avalon_lcd_16207.c** — These files implement the LCD controller device drivers for the HAL system library.

Register Map

The HAL device drivers make it unnecessary for you to access the registers directly. Therefore, Altera does not publish details about the register map. For more information, the **altera_avalon_lcd_16207_regs.h** file describes the register map, and the *Dot Matrix Character LCD Module User's Manual* from Optrex describes the register usage.

Interrupt Behavior

The LCD controller does not generate interrupts. However, the LCD driver's text scrolling feature relies on the HAL system clock driver, which uses interrupts for timing purposes.

Referenced Documents

This chapter references the *Nios II Software Developer's Handbook*.

Document Revision History

Table 8–2 shows the revision history for this chapter.

Table 8–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

9. PIO Core

Core Overview

The parallel input/output (PIO) core with Avalon® interface provides a memory-mapped interface between an Avalon® Memory-Mapped (Avalon-MM) slave port and general-purpose I/O ports. The I/O ports connect either to on-chip user logic, or to I/O pins that connect to devices external to the FPGA.

The PIO core provides easy I/O access to user logic or external devices in situations where a "bit banging" approach is sufficient. Some example uses are:

- Controlling LEDs
- Acquiring data from switches
- Controlling display devices
- Configuring and communicating with off-chip devices, such as application-specific standard products (ASSP)

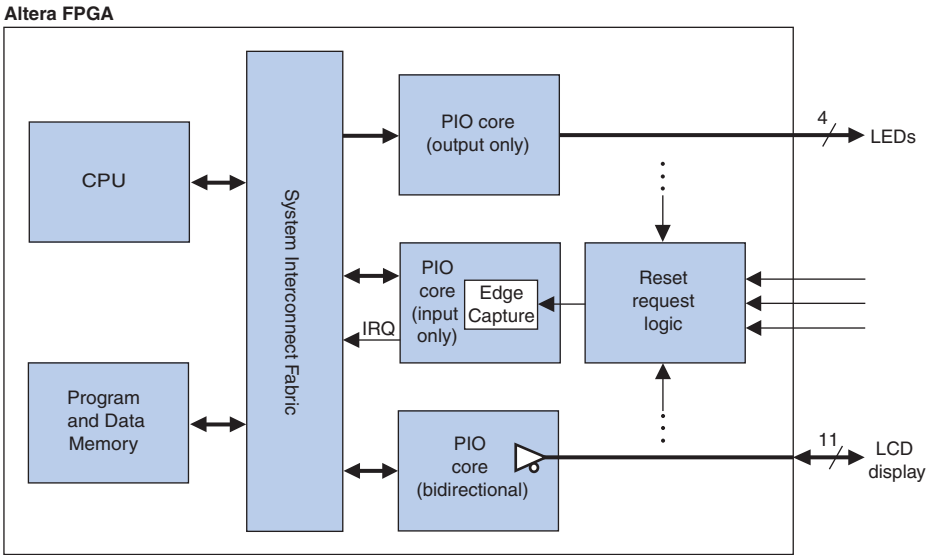
The PIO core interrupt request (IRQ) output can assert an interrupt based on input signals. The PIO core is SOPC Builder ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Example Configurations" on page 9–3
- "Instantiating the PIO Core in SOPC Builder" on page 9–4
- "Device Support" on page 9–5
- "Software Programming Model" on page 9–5

Functional Description

Each PIO core can provide up to 32 I/O ports. An intelligent host such as a microprocessor controls the PIO ports by reading and writing the register-mapped Avalon-MM interface. Under control of the host, the PIO core captures data on its inputs and drives data to its outputs. When the PIO ports are connected directly to I/O pins, the host can tristate the pins by writing control registers in the PIO core. Figure 9–1 shows an example of a processor-based system that uses multiple PIO cores to drive LEDs, capture edges from on-chip reset-request control logic, and control an off-chip LCD display.

Figure 9–1. An Example System Using Multiple PIO Cores



When integrated into an SOPC Builder-generated system, the PIO core has two user-visible features:

- A memory-mapped register space with four registers: `data`, `direction`, `interruptmask`, and `edgecapture`
- 1 to 32 I/O ports

The I/O ports can be connected to logic inside the FPGA, or to device pins that connect to off-chip devices. The registers provide an interface to the I/O ports via the Avalon-MM interface. See Table 9–2 on page 9–6 for a description of the registers.

Data Input and Output

The PIO core I/O ports can connect to either on-chip or off-chip logic. The core can be configured with inputs only, outputs only, or both inputs and outputs. If the core is used to control bidirectional I/O pins on the device, the core provides a bidirectional mode with tristate control.

The hardware logic is separate for reading and writing the data register. Reading the data register returns the value present on the input ports (if present). Writing data affects the value driven to the output ports (if present). These ports are independent; reading the data register does not return previously-written data.

Edge Capture

The PIO core can be configured to capture edges on its input ports. It can capture low-to-high transitions, high-to-low transitions, or both. Whenever an input detects an edge, the condition is indicated in the `edgecapture` register. The types of edges detected is specified at system generation time, and cannot be changed via the registers.

IRQ Generation

The PIO core can be configured to generate an IRQ on certain input conditions. The IRQ conditions can be either:

- *Level-sensitive*—The PIO core hardware can detect a high level. A `NOT` gate can be inserted external to the core to provide negative sensitivity.
- *Edge-sensitive*—The core's edge capture configuration determines which type of edge causes an IRQ

Interrupts are individually maskable for each input port. The interrupt mask determines which input port can generate interrupts.

Example Configurations

Figure 9–2 shows a block diagram of the PIO core configured with input and output ports, as well as support for IRQs.

Figure 9–2. PIO Core with Input Ports, Output Ports, and IRQ Support

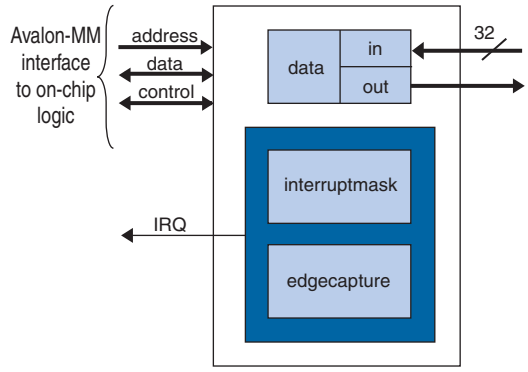
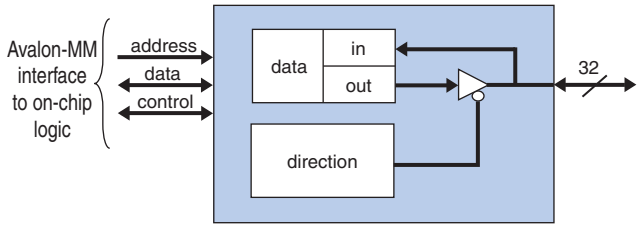


Figure 9–3 shows a block diagram of the PIO core configured in bidirectional mode, without support for IRQs.

Figure 9–3. PIO Core with Bidirectional Ports



Avalon-MM Interface

The PIO core's Avalon-MM interface consists of a single Avalon-MM slave port. The slave port is capable of fundamental Avalon-MM read and write transfers. The Avalon-MM slave port provides an IRQ output so that the core can assert interrupts.

Instantiating the PIO Core in SOPC Builder

Use the MegaWizard™ interface for the PIO core in SOPC Builder to configure the core. The following sections describe the available options.

Basic Settings

The **Basic Settings** page allows you to specify the width, direction and reset value of the I/O ports.

Width

The width of the I/O ports can be set to any integer value between 1 and 32.

Direction

You can set the port direction to one of the options shown in Table 9–1.

Table 9–1. Direction Settings

Setting	Description
Bidirectional (tristate) ports	In this mode, each PIO bit shares one device pin for driving and capturing data. The direction of each pin is individually selectable. To tristate an FPGA I/O pin, set the direction to input.
Input ports only	In this mode the PIO ports can capture input only.
Output ports only	In this mode the PIO ports can drive output only.
Both input and output ports	In this mode, the input and output ports buses are separate, unidirectional buses of *n* bits wide.

Output Port Reset Value

You can specify the reset value of the output ports. The range of legal values depends on the port width.

Output Register

The option **Enable individual bit set/clear output register** allows you to set or clear individual bits of the output port. When this option is turned on, two additional registers—`outset` and `outclear`—are implemented. You can use these registers to specify the output bit to set and clear.

Input Options

The **Input Options** page allows you to specify edge-capture and IRQ generation settings. The **Input Options** page is not available when **Output ports only** is selected on the **Basic Settings** page.

Edge Capture Register

Turn on **Synchronously capture** to include the edge capture register, `edgecapture`, in the core. The edge capture register allows the core to detect and generate an optional interrupt when an edge of the specified type occurs on an input port. The user must further specify the following features:

- Select the type of edge to detect:
  - **Rising Edge**
  - **Falling Edge**
  - **Either Edge**
- Turn on **Enable bit-clearing for edge capture register** to clear individual bit in the edge capture register. To clear a given bit, write 1 to the bit in the edge capture register.

Interrupt

Turn on **Generate IRQ** to assert an IRQ output when a specified event occurs on input ports. The user must further specify the cause of an IRQ event:

- **Level—** The core generates an IRQ whenever a specific input is high and interrupts are enabled for that input in the `interruptmask` register.
- **Edge**— The core generates an IRQ whenever a specific bit in the edge capture register is high and interrupts are enabled for that bit in the `interruptmask` register.

When **Generate IRQ** is off, the `interruptmask` register does not exist.

Simulation

The **Simulation** page allows you to specify the value of the input ports during simulation. Turn on **Hardwire PIO inputs in test bench** to set the PIO input ports to a certain value in the testbench, and specify the value in **Drive inputs to** field.

Device Support

The PIO core supports all Altera® device families.

Software Programming Model

This section describes the software programming model for the PIO core, including the register map and software constructs used to access the hardware. For Nios® II processor users, Altera provides the HAL system library header file that defines the PIO core registers. The PIO core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library.

The Nios II Embedded Design Suite (EDS) provides several example designs that demonstrate usage of the PIO core. In particular, the **count_binary.c** example uses the PIO core to drive LEDs, and detect button presses using PIO edge-detect interrupts.

Software Files

The PIO core is accompanied by one software file, **altera_avalon_pio_regs.h**. This file defines the core's register map, providing symbolic constants to access the low-level hardware.

Register Map

An Avalon-MM master peripheral, such as a CPU, controls and communicates with the PIO core via the four 32-bit registers, shown in Table 9–2. The table assumes that the PIO core's I/O ports are configured to a width of *n* bits.

Table 9–2. Register Map for the PIO Core

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	`data`	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	`direction` *(1)*		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	`interruptmask` *(1)*		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	`edgecapture` *(1)*, *(2)*		R/W	Edge detection for each input port.				
4	`outset`		W	Specifies which bit of the output port to set.				
5	`outclear`		W	Specifies which output bit to clear.				

Notes to Table 9–2:

(1) This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.

(2) Writing any value to `edgecapture` clears all bits to 0.

data Register

Reading from `data` returns the value present at the input ports. If the PIO core hardware is configured in output-only mode, reading from `data` returns an undefined value.

Writing to `data` stores the value to a register that drives the output ports. If the PIO core hardware is configured in input-only mode, writing to `data` has no effect. If the PIO core hardware is in bidirectional mode, the registered value appears on an output port only when the corresponding bit in the `direction` register is set to 1 (output).

direction Register

The `direction` register controls the data direction for each PIO port, assuming the port is bidirectional. When bit *n* in `direction` is set to 1, port *n* drives out the value in the corresponding bit of the `data` register.

The `direction` register only exists when the PIO core hardware is configured in bidirectional mode. The mode (input, output, or bidirectional) is specified at system generation time, and cannot be changed at runtime. In input-only or output-only mode, the `direction` register does not exist. In this case, reading `direction` returns an undefined value, writing `direction` has no effect.

After reset, all bits of direction are 0, so that all bidirectional I/O ports are configured as inputs. If those PIO ports are connected to device pins, the pins are held in a high-impedance state. In bi-directional mode, to change the direction of the PIO port, reprogram the `direction` register.

interruptmask Register

Setting a bit in the `interruptmask` register to 1 enables interrupts for the corresponding PIO input port. Interrupt behavior depends on the hardware configuration of the PIO core. See "Interrupt Behavior" on page 9–7.

The `interruptmask` register only exists when the hardware is configured to generate IRQs. If the core cannot generate IRQs, reading `interruptmask` returns an undefined value, and writing to `interruptmask` has no effect.

After reset, all bits of `interruptmask` are zero, so that interrupts are disabled for all PIO ports.

edgecapture Register

Bit *n* in the `edgecapture` register is set to 1 whenever an edge is detected on input port *n*. An Avalon-MM master peripheral can read the `edgecapture` register to determine if an edge has occurred on any of the PIO input ports. If the option **Enable bit-clearing for edge capture register** is turned off, writing any value to the `edgecapture` register clears all bits in the register. Otherwise, writing a 1 to a particular bit in the register clears only that bit.

The type of edge(s) to detect is fixed in hardware at system generation time. The `edgecapture` register only exists when the hardware is configured to capture edges. If the core is not configured to capture edges, reading from `edgecapture` returns an undefined value, and writing to `edgecapture` has no effect.

outset and outclear Registers

You can use the `outset` and `outclear` registers to set and clear individual bits of the output port. For example, to set bit 6 of the output port, write `0x40` to the `outset` register. Writing `0x08` to the `outclear` register clears bit 3 of the output port.

These registers are only present when the option **Enable individual bit set/clear output register** is turned on.

Interrupt Behavior

The PIO core outputs a single IRQ signal that can connect to any master peripheral in the system. The master can read either the `data` register or the `edgecapture` register to determine which input port caused the interrupt.

When the hardware is configured for level-sensitive interrupts, the IRQ is asserted whenever corresponding bits in the `data` and `interruptmask` registers are 1. When the hardware is configured for edge-sensitive interrupts, the IRQ is asserted whenever corresponding bits in the `edgecapture` and `interruptmask` registers are 1. The IRQ remains asserted until explicitly acknowledged by disabling the appropriate bit(s) in `interruptmask`, or by writing to `edgecapture`.

Software Files

The PIO core is accompanied by the following software file. This file provide low-level access to the hardware. Application developers should not modify the file.

- **altera_avalon_pio_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used by device driver functions.

Document Revision History

Table 9–3 shows the revision history for this chapter.

Table 9–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	Added a section on new registers, `outset` and `outclear`.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Added the description for **Output Port Reset Value** and **Simulation** parameters.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

10. Avalon-ST Serial Peripheral Interface Core

Core Overview

The Avalon® Streaming (Avalon-ST) Serial Peripheral Interface (SPI) core is an SPI slave that allows data transfers between SOPC Builder systems and off-chip SPI devices via Avalon-ST interfaces. Data is serially transferred on the SPI, and sent to and received from the Avalon-ST interface in bytes.

The SPI Slave to Avalon Master Bridge is an example of how this core is used. For more information on the bridge, refer to "SPI Slave/JTAG to Avalon Master Bridge Cores" on page 17–1.

The Avalon-ST Serial Peripheral Interface core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

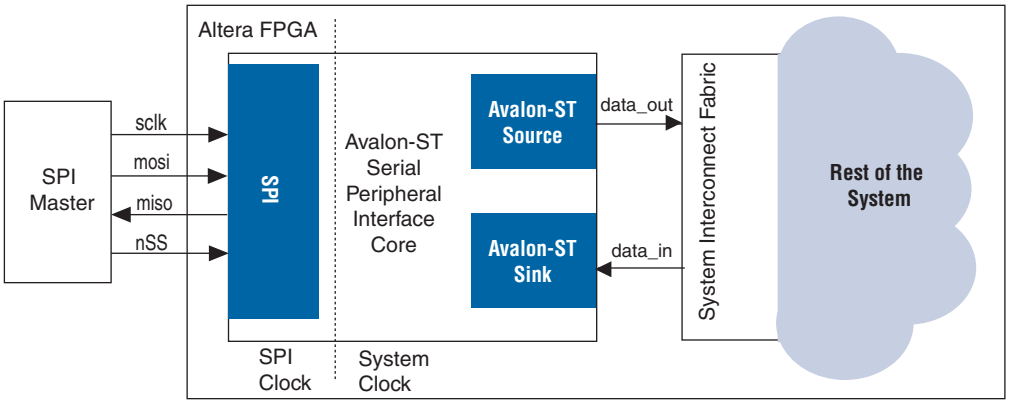
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 10–3
- "Device Support" on page 10–3

Functional Description

Figure 10–1 shows a block diagram of the Avalon-ST Serial Peripheral Interface core in a typical system configuration.

Figure 10–1. SOPC Builder System with an Avalon-ST SPI Core



Interfaces

The serial peripheral interface is full-duplex and does not support backpressure. It supports SPI clock phase bit, CPHA = 1, and SPI clock polarity bit, CPOL = 0.

Table 10–1 shows the properties of the Avalon-ST interfaces.

Table 10–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Not supported.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.
Error	Not used.
Packet	Not supported.

For more information about Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Operation

The Avalon-ST SPI core waits for the `nSS` signal to be asserted low, signifying that the SPI master is initiating a transaction. The core then starts shifting in bits from the input signal `mosi`. The core packs the bits received on the SPI to bytes and checks for the following special characters:

- `0x4a`—Idle character. The core drops the idle character.
- `0x4d`—Escape character. The core drops the escape character, and `XOR`s the following byte with `0x20`.

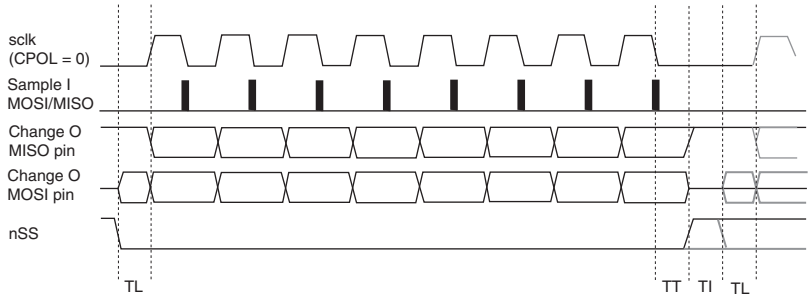
For each valid byte of data received, the core asserts the `valid` signal on its Avalon-ST source interface and presents the byte on the interface for a clock cycle.

At the same time, the core shifts data out from the Avalon-ST sink to the output signal `miso` beginning with from the most significant bit. If there is no data to shift out, the core shifts out idle characters (`0x4a`). If the data is a special character, the core inserts an escape character (`0x4d`) and XORs the data with `0x20`.

The data shifts into and out of the core in the direction of MSB first.

Figure 10–2 shows the SPI transfer protocol.

Figure 10–2. SPI Transfer Protocol



Notes to Figure 10–2:

(1) TL = The worst recovery time of `sclk` with respect with `nSS`.

(2) TT = The worst hold time for `MOSI` and `MISO` data.

(3) TI = The minimum width of a reset pulse required by Altera FPGA families.

Timing

The core requires a lead time (TL) between asserting the `nSS` signal and the SPI clock, and a lag time (TT) between the last edge of the SPI clock and deasserting the `nSS` signal. The `nSS` signal must be deasserted for a minimum idling time (TI) of one SPI clock between byte transfers. A TimeQuest SDC file (**.sdc**) is provided to remove false timing paths. The frequency of the SPI master's clock must be equal to or lower than the frequency of the core's clock.

Limitations

Daisy-chain configuration, where the output line `miso` of an instance of the core is connected to the input line `mosi` of another instance, is not supported.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST SPI core in SOPC Builder to add the core to a system. The parameter **Number of synchronizer stages: Depth** allows you to specify the length of synchronization register chains. These register chains are used when a metastable event is likely to occur and the length specified determines the meantime before failure. The register chain length, however, affects the latency of the core.

For more information on metastability in Altera devices, refer to *AN 42: Metastability in Altera Devices*. For more information on metastability analysis and synchronization register chains, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Device Support

The Avalon-ST SPI core supports all Altera® device families.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *"SPI Slave/JTAG to Avalon Master Bridge Cores" on page 17–1*
- *AN 42: Metastability in Altera Devices*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

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

Table 10–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	Added a description to specify the shift direction.	—
March 2009 v9.0.0	Added description of a new parameter, **Number of synchronizer stages: Depth**.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

11. PCI Lite Core

Core Overview

The PCI Lite core is a protocol interface that translates PCI transactions to Avalon® Memory-Mapped (Avalon-MM) transactions with low latency and high throughput. The PCI Lite core uses the PCI-Avalon bridge to connect the PCI bus to the interconnect fabric, allowing you to easily create simple PCI systems that include one or more SOPC Builder components. This core has the following features:

- SOPC Builder ready
- PCI complexities, such as retry and disconnect are handled by the PCI/Avalon Bridge logic and transparent to the user
- Run-time configurable (dynamic) Avalon-to-PCI address translation
- Separate Avalon Memory-Mapped (Avalon-MM) slave ports for PCI bus access (PBA) and control register access (CRA)
- Support for Avalon-MM burst mode
- Common PCI and Avalon clock domains
- Option to increase PCI read performance by increasing the number of pending reads and maximum read burst size.

This chapter contains the following sections:

- "Performance and Resource Utilization"
- "Functional Description" on page 11–2
- "Instantiating the Core in SOPC Builder" on page 11–11
- "Device Support" on page 11–14
- "Simulation Considerations" on page 11–14

Performance and Resource Utilization

This section lists the resource utilization and performance data for supported devices when operating in the PCI Target-Only, and PCI Master/Target device modes for each of the application-specific performance settings.

The estimates are obtained by compiling the core using the Quartus® II software. Performance results vary depending on the parameters that you specify for the system module.

Table 11–1 shows the resource utilization and performance data for a Stratix® III device (EP3SE50F780C2). The performance of the MegaCore function in the Stratix IV family is similar to the Stratix III family.

Table 11–1. Memory Utilization and Performance Data for the Stratix III Family

PCI Device Mode	PCI Target	PCI Master	ALUTs *(2)*	Logic Register	M9K Memory Blocks	I/O Pins
Min *(1)*	Enabled	N/A	715	517	2	48
Max *(1)*	Enabled	Enabled	1,347	876	5	50

Notes to Table 11–1:

(1) **Min** = One BAR with minimum settings for each parameter.
Max = Three BARs with maximum settings for each parameter.

(2) The logic element (LE) count for the Stratix III family is based on the number of adaptive look-up tables (ALUTs) used for the design as reported by the Quartus II software.

Table 11–2 lists the resource utilization and performance data for a Cyclone III device (EP3C40F780C6).

Table 11–2. Memory Utilization and Performance Data for the Cyclone III Family

PCI Device Mode	PCI Target	PCI Master	Logic Elements	Logic Register	M4K Memory Blocks	I/O Pins
Min *(1)*	Enabled	N/A	1,057	511	2	48
Max *(1)*	Enabled	Enabled	2,027	878	5	50

Note to Table 11–2:

(1) **Min** = One BAR with minimum settings for each parameter.
Max = Three BARs with maximum settings for each parameter.

Functional Description

The following sections provide a functional description of the PCI Lite Core.

PCI-Avalon Bridge Blocks

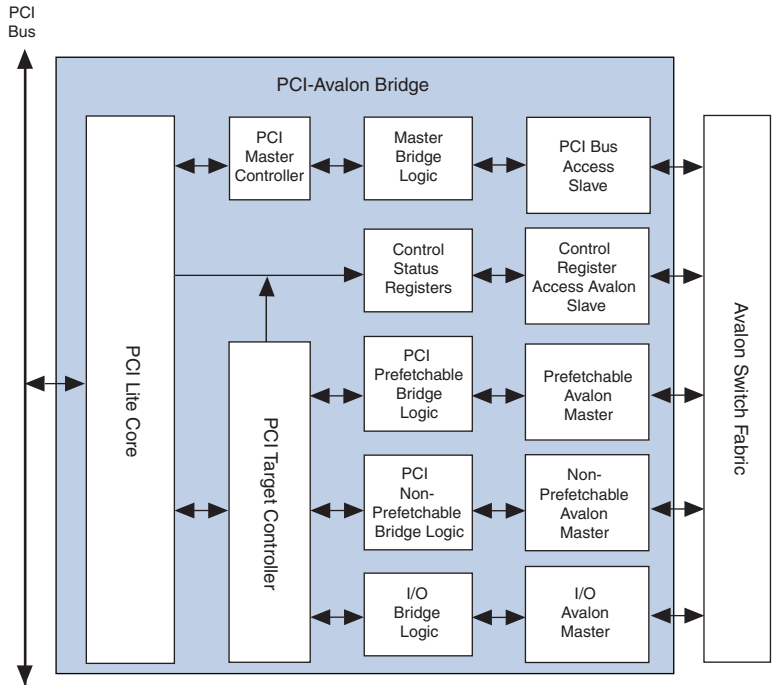
The PCI-Avalon bridge's blocks manage the connectivity for the following PCI operational modes:

- PCI Target-Only Peripheral
- PCI Master/Target Peripheral
- PCI Host-Bridge Device

Depending on the operational mode, the PCI-Avalon bridge uses some or all of the predefined Avalon-MM ports. Figure 11–1 shows a generic PCI-Avalon bridge block diagram, which includes the following blocks:

- Five predefined Avalon-MM ports
- Control registers
- PCI master controller (when applicable)
- PCI target controller

Figure 11–1. Generic PCI-Avalon Bridge Block Diagram



Avalon-MM Ports

The Avalon bridge comprises up to five predefined ports to communicate with the interconnect (depending on device operating mode).

This section discusses the five Avalon-MM ports:

- Prefetchable Avalon-MM master
- Non-Prefetchable Avalon-MM master
- I/O Avalon-MM master
- PCI bus access slave
- Control register access (CRA) Avalon-MM slave

Prefetchable Avalon-MM Master

The prefetchable Avalon-MM master port provides a high bandwidth PCI memory request access to Avalon-MM slave peripherals. This master port is capable of generating Avalon-MM burst transactions for PCI requests that hit a prefetchable base address register (BAR). You should only connect prefetchable Avalon-MM slaves to this port, typically RAM or ROM memory devices.

This port is optimized for high bandwidth transfers as a PCI target and it does not support single cycle transactions.

Non-Prefetchable Avalon-MM Master

The Non-Prefetchable Avalon-MM master port provides a low latency PCI memory request access to Avalon-MM slave peripherals. Burst operations are not supported on this master port. Only the exact amount of data needed to service the initial data phase is read from the interconnect. Therefore, the PCI byte enables (for the first data phase of the PCI read transaction) are passed directly to the interconnect.

This Avalon-MM master port is optimized for low latency access from PCI-to-Avalon-MM slaves. This is optimal for providing PCI target access to simple Avalon-MM peripherals.

I/O Avalon-MM Master

The I/O Avalon-MM master port provides a low latency PCI I/O request access to Avalon-MM slave peripherals. Burst operations are not supported on this master port. As only the exact amount of data needed to service the initial data phase is read from the interconnect, the PCI byte enables (for the first data phase of the PCI read transaction) are passed directly to the interconnect.

This Avalon-MM master port is also optimized for I/O access from PCI-to-Avalon-MM slaves for providing PCI target access to simple Avalon-MM peripherals.

PCI Bus Access Slave

This Avalon-MM slave port propagates the following transactions from the interconnect to the PCI bus:

- Single cycle memory read and write requests
- Burst memory read and write requests
- I/O read and write requests
- Configuration read and write requests

Burst requests from the interconnect are the only way to create burst transactions on the PCI bus.

This slave port is not implemented in the PCI Target-Only Peripheral mode.

Control Register Access (CRA) Avalon-MM Slave

This Avalon-MM slave port is used to access control registers in the PCI-Avalon bridge. To provide external PCI master access to these registers, one of the bridge's master ports must be connected to this port. There is no internal access inside the bridge from the PCI bus to these registers. You can only write to these registers from the interconnect. The `Control Register Access Avalon Slave` port is only enabled on Master/Target selection. The range of values supported by PCI CRA is 0x1000 to 0x1FFF. Depending on the system design, these values can be accessed by PCI processors, Avalon processors or both.

Table 11–3 on page 11–5 shows the instructions on how to use these values. The address translation table is writable via the `Control Register Access Avalon Slave` port. If the **Number of Address Pages** field is set to the maximum of **512**, `0x1FF8` contains `A2P_ADDR_MAP_LO511` and `0x1FFC` contains `A2P_ADDR_MAP_HI511`.

Each entry in the PCI address translation table is always 8 bytes wide. The lower order address bits that are treated as a pass through between Avalon-MM and PCI, and the number of pass-through bits, are defined by the size of page in the address translation table and are always forced to `0` in the hardware table. For example, if the page size is 4 KBytes, the number of pass-through bits is
$\log_2$ (page size) = $\log_2$ (4 KBytes) = 12.

Refer to "Avalon-to-PCI Address Translation" on page 11–6 for more details.

Table 11–3. Avalon-to-PCI Address Translation Table – Address Range: 0x1000-0x1FFF

Address	Bit	Name	Access Mode	Description
`0x1000`	1:0	`A2P_ADDR_SPACE0`	W	Address space indication for entry 0. See Table 11–4 on page 11–7 for the definition of these bits.
	31:2	`A2P_ADDR_MAP_LO0`	W	Lower bits of Avalon-to-PCI address map entry 0. The pass through bits are not writable and are forced to 0.
`0x1004`	31:0	`A2P_ADDR_MAP_HI0`	W	Reserved.
`0x1008`	1:0	`A2P_ADDR_SPACE1`	W	Address Space indication for entry 1. See Table 11–4 on page 11–7 for the definition of these bits.
	31:2	`A2P_ADDR_MAP_LO1`	W	Lower bits of Avalon-to-PCI address map entry 1. Pass through bits are not writable and are forced to 0. This entry is only implemented if the number of pages in the address translation table is greater than 1.
`0x100C`	31:0	`A2P_ADDR_MAP_HI1`	W	Reserved.

Master and Target Performance

The performance of the PCI Lite core is designed to provide low-latency single-cycle and burst transactions.

Master Performance

The master provides high throughput for transactions initiated by Avalon-MM master devices to PCI target devices via the PCI bus master interface. Avalon-MM read transactions are implemented as latent read transfers. The PCI master device issues only one read transaction at a time.

☞ The PCI bus access (PBA) handles the Avalon master transaction system interconnect hold state for 6 clock cycles. This is the maximum number of cycles supported by the PCI specification.

Target Performance

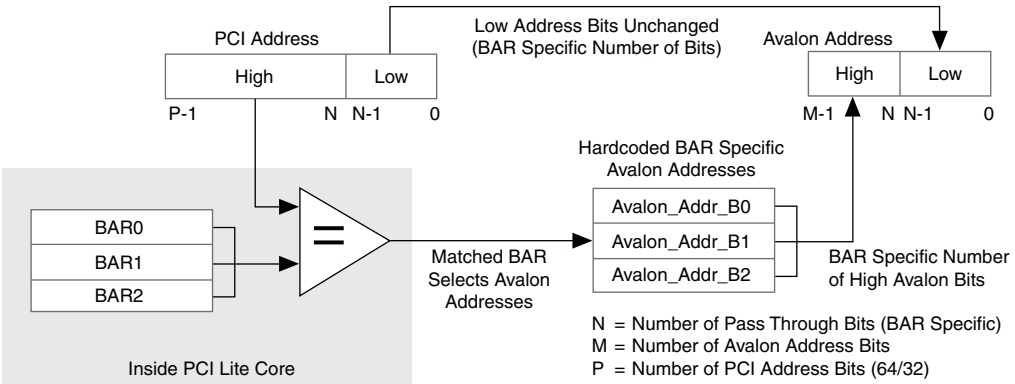
The target allows high throughput read/write operations to Avalon-MM slave peripherals. Read/write accesses to prefetchable base address registers (BARs) use dual-port buffers to enable burst transactions on both the PCI and Avalon-MM sides. This profile also allows access to the PCI BARs (Prefetchable, Non-Prefetchable, and I/O) to use their respective Avalon-MM master ports to initiate transfers to Avalon-MM slave peripherals. Prefetchable handles burst transaction and Non-Prefetchable and I/O handles only single-cycle transaction.

All PCI read transactions are completed as delayed reads. However, only one delayed read is accepted and processed at a time.

PCI-to-Avalon Address Translation

Figure 11–2 shows the PCI-to-Avalon address translation. The bits in the PCI address that are used in the BAR matching process are replaced by an Avalon-MM base address that is specific to that BAR.

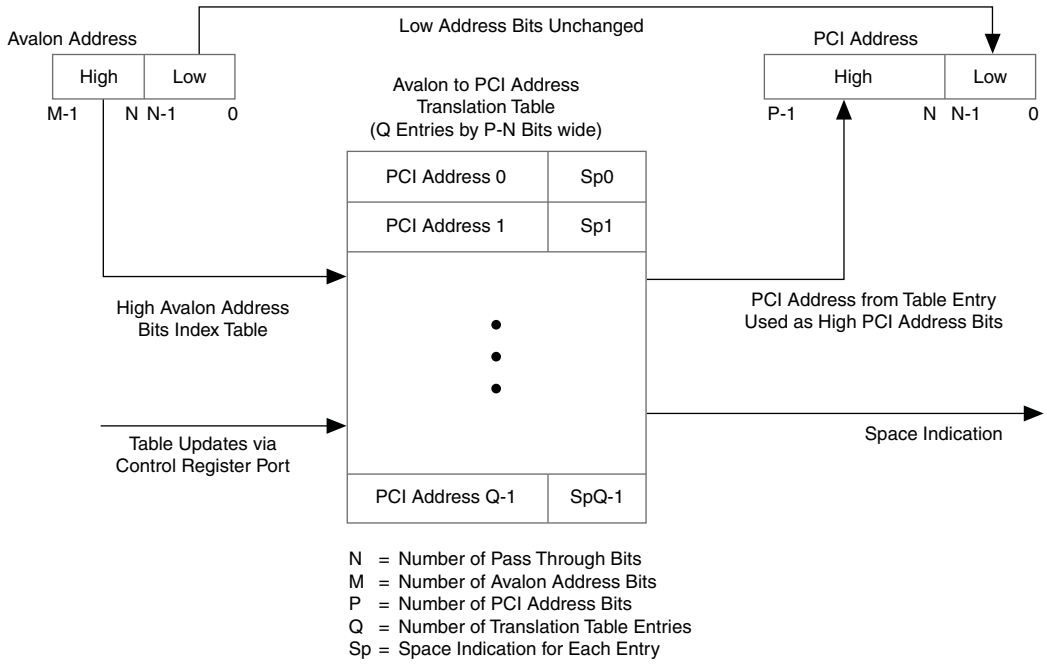
Figure 11–2. PCI-to-Avalon Address Translation



Avalon-to-PCI Address Translation

Avalon-to-PCI address translation is done through a translation table. Low order Avalon-MM address bits are passed to PCI unchanged; higher order Avalon-MM address bits are used to index into the address translation table. The value found in the table entry is used as the higher order PCI address bits. Figure 11–3 depicts this process.

Figure 11–3. Avalon-to-PCI Address Translation



The address size selections in the translation table determine both the number of entries in the Avalon-to-PCI address translation table, and the number of bits that are passed through the transaction table unchanged.

Each entry in the address translation table also has two address space indication bits, which specify the type of address space being mapped. If the type of address space being mapped is memory, the bits also indicate the resulting PCI address is a 32-bit address.

Table 11–4 shows the address space field's format of the address translation table entries.

Table 11–4. Address Space Bit Encodings

Address Space Indicator (Bits 1:0)	Description
00	Memory space, 32-bit PCI address. Address bits 63:32 of the translation table entries are ignored.
01	Reserved.
10	I/O space. The address from the translation table process is modified as described in Table 11–5.
11	Configuration space. The address from the translation table process is treated as a type 1 configuration address and is modified as described in Table 11–5.

If the space indication bits specify configuration or I/O space, subsequent modifications to the PCI address are performed. See Table 11–5.

Table 11–5. Configuration and I/O Space Address Modifications

Address Space	Modifications Performed
I/O	■ Address bits 2:0 are set to point to the first enabled byte according to the Avalon byte enables. (Bit 2 only needs to be modified when a 64-bit data path is in use.) ■ Address bits 31:3 are handled normally.
Configuration address bits 23:16 == 0 (bus number == 0)	■ Address bits 1:0 are set to `00` to indicate a type 0 configuration request. ■ Address bits 10:2 are passed through as normal. ■ Address bits 31:11 are set to be a one-hot encoding of the device number field (15:11) of the address from the translation table. For example, if the device number is `0x00`, address bit 11 is set to 1 and bits 31:12 are set to 0. If the device number is `0x01`, address bit 12 is set to 1 and bits 31:13, 11 are set to 0. ■ Address bits 31:24 of the original PCI address are ignored.
Configuration address bits 23:16 > 0 (bus number > 0)	■ Address bits 1:0 are set to `01` to indicate a type 1 configuration request. ■ Address bits 31:2 are passed through unchanged.

Avalon-To-PCI Read and Write Operation

The PCI Bus Access Slave port is a burst-capable slave that attempts to create PCI bursts that match the bursts requested from the interconnect.

The PCI-Avalon bridge is capable of handling bursts up to 512 bytes with a 32-bit PCI bus. In other words, the maximum supported Avalon-MM burst count is 128.

Bursts from Avalon-MM can be received on any boundary. However, when internal PCI-Avalon bridge bursts cross the Avalon-to-PCI address page boundary, they are broken into two pieces. Two bursts are used because the address translation can change at that boundary, requiring a different PCI address for the second portion of the burst with a burst count greater than 1.

☞ Avalon-MM burst read requests are treated as if they are going to prefetchable PCI space. Therefore, if the PCI target space is non-prefetchable, you should not use read bursts.

Several factors control how Avalon-MM transactions (bursts or single cycle) are translated to PCI transactions. These cases are discussed in Table 11–6.

Table 11–6. Translation of Avalon Requests to PCI Requests

Data Path Width	Avalon Burst Count	Type of Operation	Avalon Byte Enables	Resulting PCI Operation and Byte Enables
32	1	Read or Write	Any value	Single data phase read or write, PCI byte enables identical to Avalon byte enables
32	>1	Read	Any value	Attempt to burst on PCI. All data phases have all PCI bytes enabled.
32	>1	Write	Any value	Attempt to burst on PCI. All data phases have PCI byte enables identical to the Avalon byte enables.

Avalon-to-PCI Write Requests

For write requests from the interconnect, the write request is pushed onto the PCI bus as a configuration write, I/O write, or memory write. When the Avalon-to-PCI command/write data buffer either has enough data to complete the full burst or 8 data phases (32 bytes on a 32-bit PCI bus) are exceeded, the PCI master controller issues the PCI write transaction.

The PCI write is issued to configuration, I/O, or memory space based on the address translation table. See "Avalon-to-PCI Address Translation" on page 11–6.

A PCI write burst can be terminated for various reasons. Table 11–7 describes the resulting action for the PCI master write request termination condition.

Table 11–7. PCI Master Write Request Termination Conditions

Termination condition	Resulting Action
Burst count satisfied	Normal master-initiated termination on PCI bus, command completes, and the master controller proceeds to the next command.
Latency timer expiring during configuration, I/O, or memory write command	Normal master-initiated termination on PCI bus, the continuation of the PCI write is requested from the master controller arbiter.
Avalon-to-PCI command/write data buffer running out of data	Normal master-initiated termination on the PCI bus. Master controller waits for the buffer to reach 8 DWORDs on a 32-bit PCI or 16 DWORDs on a 64-bit PCI, or there is enough data to complete the remaining burst count. Once enough data is available, the master controller arbiter continues wth the PCI write.
PCI target disconnect	The master controller arbiter attempts to initiate the PCI write until the transaction is successful.
PCI target retry	
PCI target-abort	The rest of the write data is read from the buffer and discarded.
PCI master-abort	

Avalon-to-PCI Read Requests

For read requests from the interconnect, the request is pushed on the PCI bus by a configuration read, I/O read, memory read, memory read line, or memory read multiple command. The PCI read is issued to configuration, I/O, or memory space based on the address translation table entry. See "Avalon-to-PCI Address Translation" on page 11–6.

If a memory space read request can be completed in a single data phase, it is issued as a memory read command. If the memory space read request spans more than one data phase but does not cross a cacheline boundary (as defined by the cacheline size register), it is issued as a memory read line command. If the memory space read request crosses a cache line boundary, it is issued as multiple memory read commands.

Read requests on PCI may initially be retried. Retries depend on the response time from the target. The master continues to retry until it gets the required data.

Table 11–8 shows PCI master read request termination conditions.

Table 11–8. PCI Master Read Request Termination Conditions

Termination Condition	Resulting Action
Burst count satisfied	Normal master initiated termination on the PCI bus. Master controller proceeds to the next command.
Latency timer expired	Normal master initiated termination on PCI bus. The continuation of the PCI read is made pending as a request from the master controller arbiter.
PCI target disconnect	The continuation of the PCI read is requested from the master controller arbiter.
PCI target retry	
PCI target-abort	Dummy data is returned to complete the Avalon-MM read request. The next operation is then attempted in a normal fashion.
PCI master-abort	

Ordering of Requests

The PCI-Avalon bridge handles the following types of requests:

- PMW—Posted memory write.
- DRR—Delayed read request.
- DWR—Delayed write request. DWRs are I/O or configuration write operation requests. The PCI-Avalon bridge does not handle DWRs as delayed writes.
  - As a PCI master, I/O or configuration writes are generated from posted Avalon-MM writes. If required to verify completion, you must issue a subsequent read request to the same target.
  - As a PCI target, configuration writes are the only requests accepted, which are never delayed. These requests are handled directly by the PCI core.
- DRC—Delayed read completion.
- DWC—Delayed write completion. These are never passed through to the core in either direction. Incoming configuration writes are never delayed. Delayed write completion status is not passed back at all.

Every single transaction that is initiated, locks the core until it is completed. Only then can a new transaction be accepted.

PCI Interrupt

When Avalon-MM asserts the `IRQ` signal, an interrupt on the PCI bus occurs. The Avalon-MM `IRQ` input causes a bit to be set in the PCI interrupt status register.

Instantiating the Core in SOPC Builder

Table 11–9 describes the parameters that can be configured in SOPC Builder for the PCI Lite core.

Table 11–9. Parameters for PCI Lite Core (Part 1 of 2)

Parameters	Legal Values	Description
Enable Master/Target Mode	On or Off	Turning this option **On** enables Master/Target mode. This option enables allows Avalon-MM master devices to access PCI target devices via the PCI bus master interface, and PCI bus master devices to access Avalon-MM slave devices via the PCI bus target interface. Turning this option **Off** means you have selected Target Only mode, which allows PCI bus mastering devices to access Avalon-MM slave devices via the PCI bus target interface.
Enable Host Bridge Mode	On or Off	Turning this option **On** enables this mode. In addition to the same features provided by the PCI Master/Target mode, Host Bridge Mode provides host bridge functionality including hardwiring the master enable bit to 1 in the PCI command register and allowing self-configuration. This value can only be set if the **Enable Master/Target Mode** option is turned **On**.
Number of Address Pages	2, 4, 8, or 16	The number of translation/pages supported by the device for Avalon to PCI address translation.
Size of Address Pages	12–27	The supported address size (in bits) that can be assigned to each map number entries.
Prefetchable BAR	On or Off	Turning this option **On** invokes a Prefetchable Master (PM) Bar in the PCI system. This option allows PCI-Avalon Bridge Lite to accept and process PM transactions.
Prefetchable BAR Size	10–31	The allowed reserved address range supported by the PM BAR. The reserved memory space is 1 KByte (10 bits) to 4 GBytes (31 bits).
Prefetchable BAR Avalon Address Translation Offset	*<BAR translation value>*	The direct translation of the value that hits the BAR and modified to a fixed address in the Avalon space. Refer to "PCI-to-Avalon Address Translation" on page 11–6.
Non-Prefetchable BAR	On or Off	Turning this option **On** invokes a Non-Prefetchable Master (NPM) Bar in the PCI system. This option allows the PCI-Avalon Bridge Lite to accept and process NPM transactions.
Non-Prefetchable BAR Size	10–31	Specifies the allowed reserved address range supported by the NPM BAR. The reserved memory space is 1 KByte (10 bits) to 4 GBytes (31 bits).
Non-Prefetchable BAR Avalon Address Translation Offset	*<BAR translation value>*	The direct translation of the value that hits the BAR and modified to a fixed address in the Avalon space. Refer to "PCI-to-Avalon Address Translation" on page 11–6.
I/O BAR	On or Off	Turning this option **On** enables an I/O BAR in the system. This option allows PCI-Avalon Bridge Lite to accept and process I/O type transactions.
I/O BAR Size	2–8	The allowed reserved address range supported by the I/O BAR. The reserved memory space is 4 bytes (2 bits) to 256 bytes (8 bits).
I/O BAR Avalon Address Translation Offset	*<BAR translation value>*	The direct translation of the value that hits the BAR and modified to a fixed address in the Avalon address space. Refer to "PCI-to-Avalon Address Translation" on page 11–6.

Table 11–9. Parameters for PCI Lite Core (Part 2 of 2)

Parameters	Legal Values	Description
Maximum Target Read Burst Size	1, 2, 4, 8, 16, 32, 64, or 128	Specifies the maximum FIFO depth that is used for reading. Larger values allow more reads to be read in a single transaction but also require more time to clear the FIFO content.
Device ID	*<register value>*	Device ID register. This parameter is a 16-bit hexadecimal value that sets the device ID register in the configuration space.
Vendor ID	*<register value>*	Vendor ID register.This parameter is a 16-bit read-only register that identifies the manufacturer of the device. The value of this register is assigned by the PCI Special Interest Group (SIG).
Class Code	*<register value>*	Class code register. This parameter is a 24-bit hexadecimal value that sets the class code register in the configuration space. The value entered for this parameter must be valid PCI SIG-assigned class code register value.
Revision ID	*<register value>*	Revision ID register. This parameter is an 8-bit read-only register that identifies the revision number of the device. The value of this register is assigned by the manufacturer.
Subsystem ID	*<register value>*	Subsystem ID register. This parameter is a 16-bit hexadecimal value that sets the subsystem ID register in the PCI configuration space. Any value can be entered for this parameter.
Subsystem Vendor ID	*<register value>*	Subsystem vendor ID register. This parameter is a 16-bit hexadecimal value that sets the subsystem vendor ID register in the PCI configuration space. The value for this parameter must be a valid PCI SIG-assigned vendor ID number.
Maximum Latency	*<register value>*	Maximum latency register. This parameter is an 8-bit hexadecimal value that sets the maximum latency register in the configuration space. This parameter must be set according to the guidelines in the PCI specifications. Only meaningful when the **Enable Master/Target Mode** option is turned **On**.
Minimum Grant	*<register value>*	Minimum grant register. This parameter is an 8-bit hexadecimal value that sets the minimum grant register in the PCI configuration space. This parameter must be set according to the guidelines in the PCI specifications. Only meaningful when the **Enable Master/Target Mode** option is turned **On**.

PCI Timing Constraint Files

The PCI Lite core supplies a Tcl timing constraint file for your target device family.

When run, the constraint file automatically sets the PCI Lite core assignments for your design such as PCI Lite core hierarchy, device family, density and package type used in your Quartus II project.

To run a PCI constraint file, perform the following steps:

1. Copy **pci_constraints.tcl** from **<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite**.

2. Update the pin list in the Tcl constraint file. Edit the `get_user_pin_name` procedure in the Tcl constraint file to match the default pin names. To edit the PCI constraint file, follow these steps:

   a. Locate the `get_user_pin_name` procedure. This procedure maps the default PCI pin names to user PCI pin names. The following lines are the first few lines of the procedure:

```
proc get_user_pin_name { internal_pin_name } {

    #---------------- Do NOT change ------------------------------ ---- Change -----
    array set map_user_pin_name_to_internal_pin_name {ad                    ad      }
```

   b. Edit the pin names under the `Change` header in the file to match the PCI pin names used in your Quartus II project. In the following example, the name `ad` is changed to `pci_ad`:

```
   #---------------- Do NOT change ------------------------------     ---- Change -----
   array set map_user_pin_name_to_internal_pin_name { ad                pci_ad        }
```

   The Tcl constraint file uses the default PCI pin names to make assignments. When overwriting existing assignments, the Tcl constraint file checks the new assignment pin names against the default PCI pin names. You must update the assignment pin names if there is a mismatch between the assignment pin names and the default PCI pin names.

3. Source the constraint file by typing the following in the Quartus II Tcl Console window:

   `source pci_constraints.tcl` ↵

4. Add the PCI constraints to your project by typing the following command in the Quartus II Tcl Console window:

   `add_pci_constraints` ↵

   See "Additional Tcl Option" on page 11–13 for the option supported by the **add_pci_constraints** command.

   When you add the timing constraints file as described in Step 4 above, the Quartus II software generates a Synopsys Design Constraints (**.sdc**) file with the file name format, *<variation name>***.sdc**. The Quartus II TimeQuest timing analyzer uses the constraints specified in this file.

   For more information about **.sdc** files or TimeQuest timing analyzer, refer to Quartus II Help.

Additional Tcl Option

If you do not want to compile your project and prefer to skip analysis and synthesis, you can use the `-no_compile` option:

```
add_pci_contraints [-no_compile]
```

By default, the `add_pci_constraints` command performs analysis and synthesis in the Quartus II software to determine the hierarchy of your PCI Lite core design. You should only use this option if you have already performed analysis and synthesis or fully compiled your project prior to using this script.

Device Support

The PCI Lite core supports the Arria® GX, Arria II, Cyclone® III, Hardcopy® II, Stratix® III, and Stratix IV device families.

Simulation Considerations

The PCI Lite core includes a testbench that facilitates the design and verification of systems that implement the Altera PCI-Avalon bridge. The testbench only works for master systems and is provided in Verilog HDL only.

To use the PCI testbench, you must have a basic understanding of PCI bus architecture and operations.This section describes the features and applications of the PCI testbench to help you successfully design and verify your design.

Features

The PCI testbench includes the following features:

- Easy to use simulation environment for any standard Verilog HDL simulator
- Open source Verilog HDL files
- Flexible PCI bus functional model to verify your application that uses any PCI Lite core
- Simulates all basic PCI transactions including memory read/write operations, I/O read/write transactions, and configuration read/write transactions
- Simulates all abnormal PCI transaction terminations including target retry, target disconnect, target abort, and master abort
- Simulates PCI bus parking

Master Transactor (mstr_tranx)

The master transactor simulates the master behavior on the PCI bus. It serves as an initiator of PCI transactions for PCI testbench. The master transactor has three main sections:

- TASKS (Verilog HDL)
- INITIALIZATION
- USER COMMANDS

TASKS Sections

The TASKS (Verilog HDL) sections define the events that are executed for the user commands supported by the master transactor. The events written in the TASKS sections follow the phases of a standard PCI transaction as defined by the *PCI Local Bus Specification, Revision 3.0*, including:

- Address phase
- Turn-around phase (read transactions)
- Data phases
- Turn-around phase

The master transactor terminates the PCI transactions in the following cases:

- The PCI transaction has successfully transferred all the intended data.
- The PCI target terminates the transaction prematurely with a target retry, disconnect, or abort as defined in the *PCI Local Bus Specification, Revision 3.0*.
- A target does not claim the transaction resulting in a master abort.

The bus monitor informs the master transactor of a successful data transaction or a target termination. Refer to the source code, which shows you how the master transactor uses these termination signals from the bus monitor.

The PCI testbench master transactor TASKS sections implement basic PCI transaction functionality. If your application requires different functionality, modify the events to change the behavior of the master transactor. Additionally, you can create new procedures or tasks in the master transactor by using the existing events as an example.

INITIALIZATION Section

This user-defined section defines the parameters and reset length of your PCI bus on power-up. Specifically, the system should reset the bus and write the configuration space of the PCI agents. You can modify the master transactor INITIALIZATION section to match your system requirements by changing the time that the system reset is asserted and by modifying the data written in the configuration space of the PCI agents.

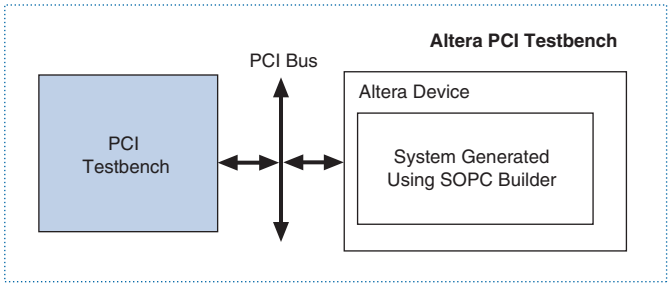
USER COMMANDS Section

The master transactor USER COMMANDS section contains the commands that initiate the PCI transactions you want to run for your tests. The list of events that are executed by these commands is defined in the TASKS sections. Customize the USER COMMANDS section to execute the sequence of commands needed to test your design.

Simulation Flow

This section describes the simulation flow using Altera PCI testbench. Figure 11–4 shows the block diagram of a typical verification environment using the PCI testbench.

Figure 11–4. Typical Verification Environment Using the PCI Testbench



The simulation flow using Altera PCI testbench comprises the following steps.

1. Use SOPC Builder to create your system. SOPC creates the *<variation name*_**system>_sim** folder in your project directory.
2. Source **pci_constraints.tcl**.
3. Copy **<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite/trgt_tranx_mem_init.dat** to **<project_directory>/<variation name_system>_sim** folder.
4. Edit the top level HDL verilog files in the testbench. Insert the following lines just before `module test_bench`.

```
`include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/pci_tb.v"

`include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/clk_gen.v"

`include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/arbiter.v"

`include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/pull_up.v"

`include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/monitor.v"

`include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/trgt_tranx.v"

`include "mstr_tranx.v"
```

> Modify **mstr_tranx.v** in your project directory to add the PCI transactions to your system. If you regenerate your system, SOPC Builder overwrites the testbench files in the **<sopc_system>_sim** directory. If you want the default testbench files, regenerate the system. Then resource **pci_constraints.tcl** or simply copy the **mstr_tranx.v** from **<quartus_ip>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite** into your project folder and repeat steps 3 and 4.

5. Set the initialization parameters, which are defined in the master transactor model source code. These parameters control the address space reserved by the target transactor model and other PCI agents on the PCI bus.
6. The master transactor defines the tasks (Verilog HDL) needed to initiate PCI transactions in your testbench. Add the commands that correspond to the transactions you want to implement in your tests to the master transactor model source code. At a minimum, you must add configuration commands to set the BAR for the target transactor model and write the configuration space of the PCI Lite core. Additionally, you can add commands to initiate memory or I/O transactions to the PCI Lite core.

7. Compile the files in your simulator, including the testbench modules and the files created by SOPC Builder.
8. Simulate the testbench for the desired time period.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 11–10 shows the revision history for this chapter.

Table 11–10. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Edited the command errors in the **Simulation Flow** section.	—
May 2008 v8.0.0	Initial release.	—

12. Cyclone III Remote Update Controller Core

Core Overview

The Cyclone® III Remote Update Controller core provides a method to control the Cyclone III remote update block from SOPC Builder systems. The core allows you to access all features of the ALTREMOTE_UPDATE megafunction through a simple Avalon® Memory-Mapped (Avalon-MM) slave interface. The slave interface allows Avalon-MM master peripherals, such as a Nios® II processor, to communicate with the core simply by reading and writing the registers.

The Cyclone III Remote Update Controller core is a thin Avalon interface layer on top of the ALTREMOTE_UPDATE megafunction. Every function of the core maps directly to a function of the megafunction. Altera recommends that you familiarize yourself with the ALTREMOTE_UPDATE megafunction before using the core.

For more information about the ALTREMOTE_UPDATE megafunction, refer to the *altremote_update Megafunction User Guide*. For more information about remote system upgrade in Cyclone III devices, refer to the *Remote System Upgrade With Cyclone III Devices* chapter in volume 1 of the *Cyclone III Device Handbook*.

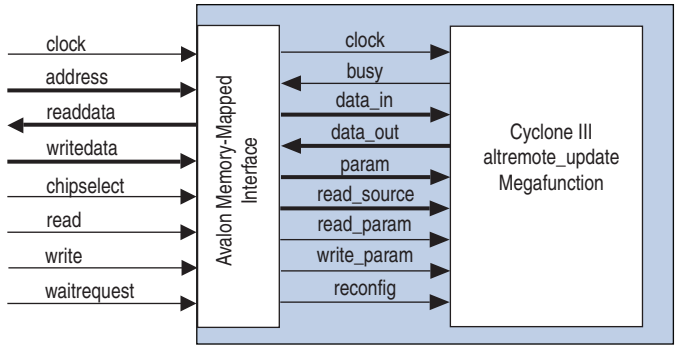
The Cyclone III Remote Update Controller core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 12–2
- "Instantiating the Core in SOPC Builder" on page 12–2

Functional Description

Figure 12–1 shows a block diagram of the Cyclone III Remote Update Controller core.

Figure 12–1. Cyclone III Remote Update Controller Core Block Diagram



Avalon-MM Slave Interface and Registers

The `address` bus on the core's Avalon-MM interface is 6 bits wide. The lower three bits of the `address` bus map directly to the `param` signal of the ALTREMOTE_UPDATE megafunction whereas the upper three bits map to the `read_source` signal.

Reading or writing to address offsets 0x00 – 0x1F of the Cyclone III Remote Update Controller core is equivalent to performing read or write operations to the ALTREMOTE_UPDATE megafunction using the `param` and `read_source` signals.

Table 12–1 shows the mapping of the 5 lowest order Remote Update Controller address bits to the ALTREMOTE_UPDATE megafunction signals.

Table 12–1. Avalon-MM Address Bits to Megafunction Signals Mapping

Address Bit	Megafunction Signal
address[0]	param[0]
address[1]	param[1]
address[2]	param[2]
address[3]	read_source[0]
address[4]	read_source[1]

The highest order address bit [5] is used to access a single `control/status` register. Reading or writing any address offset from 0x20 – 0x3F accesses the `control/status` register.

Table 12–2 shows the bit map of the `control/status` register.

Table 12–2. Bit Map of Control/Status Register

Bit(s)	Field	Access	Description
0	RECONFIG	RW	Set this bit to 1 to reset the FPGA and trigger reconfiguration.
1	RESET_TIMER	RW	Set this bit to 1 to reset the watchdog timer. Then, set this bit to 0 to allow the watchdog timer to continue.
2..31	Reserved		

Device Support

The Cyclone III Remote Update Controller core can only target Cyclone III device family. Both CFI flash and EPCS configuration devices are supported as non-volatile storage for configuration images.

Instantiating the Core in SOPC Builder

The Cyclone III Remote Update Controller core has no user-configurable parameters.

Software Programming Model

Software programs can operate the Cyclone III Remote Update Controller core by reading from and writing to the core's registers.

You can only reconfigure the FPGA to an application image from the factory image. Any attempt to reconfigure from an already reconfigured application image causes the FPGA to return to the factory image.

This section describes the most common types of operations using the Cyclone III Remote Update Controller core.

Setting the Configuration Offset

Before you reconfigure the FPGA, you must first specify the offset within the memory device from which you want to execute a reconfiguration. The offset is the relative address within the memory device where the configuration image is located. Write the offset value to address 0x04 of the core to set the configuration offset.

For example, if your system contains a CFI flash memory mapped at address 0x04000000, and the configuration image is located at address 0x100000 in the flash memory, the offset to set in the Cyclone III Remote Update Controller core is 0x100000.

Shifting the Configuration Offset Value

The ALTREMOTE_UPDATE megafunction requires that you provide only the 22 highest-order bits of a 24-bit address offset. To translate the address, right shift the offset by two bits. This results in a properly oriented 22-bit address offset.

If you are using a CFI flash device, you must also take into account the data width of the flash. If the data width of your flash device is 16 bits, you must provide a 16-bit address offset to the Cyclone III Remote Update Controller core. This requires an additional 1-bit right shift of the byte address offset. No translation is necessary if the data width of your flash is 8 bits.

If you are using an EPCS serial configuration device, consider the data width of the device to be 8 bits. Even though the EPCS device is a serial device, it uses byte addressing internally.

For example, an FPGA is set up to configure itself using active parallel mode from a 16-bit CFI flash memory mapped at address 0x04000000 in an SOPC Builder system, and the configuration image is located at byte offset 0x100000 within the flash memory. To derive the correct configuration offset, you must first right shift the byte offset 0x100000 by one bit to obtain the 16-bit address. Then, right shift by another two bits to obtain the highest 22 bits of the 24-bit offset. The result is a configuration offset of 0x20000 (0x100000 >> 3 = 0x20000), to be written to address 0x04 of the core.

Setting up the Watchdog Timer

You can set up the watchdog timer by writing the upper 12 bits of the 29-bit timeout value to address 0x02 of the core. To reset the watchdog timer, set the `RESET_TIMER` bit of the `control/status` register to 1 and immediately set the bit to 0.

CAUTION Ensure that you don't accidentally set bit 0 of the `control/status` register to 1. Otherwise, you will trigger a reconfiguration of the FPGA.

For more information on watchdog timer, refer to the *ALTREMOTE_UPDATE Megafunction User Guide*.

If you do not use the watchdog timer feature of the ALTREMOTE_UPDATE megafunction, it must be disabled before a reconfiguration is performed. To disable the watchdog timer, write `0x00` to address 0x03 of the core.

Triggering a Reconfiguration

You can trigger a reconfiguration once you have set the reconfiguration offset in the Cyclone III Remote Update Controller core, and you have either setup or disabled the watchdog timer. To trigger a reconfiguration, set the `RECONFIG` bit in the `control/status` register to 1. Consequently, the FPGA performs a reset and reconfigures itself from the configuration image specified.

Code Example

Example 12–1 shows a C function that can be used to operate the Cyclone III Remote Update Controller core from a processor such as Nios II.

Example 12–1. FPGA Reconfiguration Function

```
/*****************************************************************************
 * Function: CycloneIII_Reconfig
 * Purpose: Uses the ALT_REMOTE_UPDATE megafunction to reconfigure a Cyclone III FPGA.
 * Parameters:
 *    remote_update_base -  base address of the remote update controller
 *    flash_base         -  base address of flash device
 *    reconfig_offset    -  offset in flash from which to reconfigure
 *    watchdog_timeout   -  29-bit watchdog timeout value
 *    width_of_flash     -  data-width of flash device
 * Returns: 0 ( but never exits since it reconfigures the FPGA )
 ****************************************************************************/
int CycloneIII_Reconfig( int remote_update_base,
                         int flash_base,
                         int reconfig_offset,
                         int watchdog_timeout,
                         int width_of_flash )
{int offset_shift;

  // Obtain upper 12 bits of 29-bit watchdog timeout value
  watchdog_timeout = watchdog_timeout >> 17;

  // Only enable the watchdog timer if its timeout value is greater than 0.
  if( watchdog_timeout > 0 )
  {
    // Set the watchdog timeout value
    IOWR( remote_update_base, 0x2, watchdog_timeout );
  }
  else
  {
   // Disable the watchdog timer
   IOWR( remote_update_base, 0x3, 0 );
  }

  // Calculate how much to shift the reconfig offset location:
  //  width_of_flash == 8->offset_shift = 2.
  //  width_of_flash == 16->offset_shift = 3
  offset_shift = (( width_of_flash / 8 ) + 1 );

  // Write the offset of the desired reconfiguration image in flash
  IOWR( remote_update_base, 0x4, reconfig_offset >> offset_shift );

  // Perform the reconfiguration by setting bit 0 in the
  // control/status register
  IOWR( remote_update_base, 0x20, 0x1 );

  return( 0 );
}
```

Related Documentation

This chapter references the following documents:

- *altremote_update Megafunction User Guide*
- *Remote System Upgrade With Cyclone III Devices* in volume 1 of the *Cyclone III Device Handbook*.

Document Revision History

Table 12–3 shows the revision history for this chapter.

Table 12–3. Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Initial release.	—

13. MDIO Core

Core Overview

The Altera Management Data Input/Output (MDIO) IP core is a two-wire standard management interface that implements a standardized method to access the external Ethernet PHY device management registers for configuration and management purposes. The MDIO IP core is IEEE 802.3 standard compliant. To access each PHY device, the PHY register address must be written to the register space followed by the transaction data. The PHY register addresses are mapped in the MDIO core's register space and can be accessed by the host processor via the Avalon® Memory-Mapped (Avalon-MM) interface. This IP core can also be used with the Altera 10-Gbps Ethernet MAC to realize a fully manageable system.

The MDIO core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 13–4
- "Register Map" on page 13–5

Functional Description

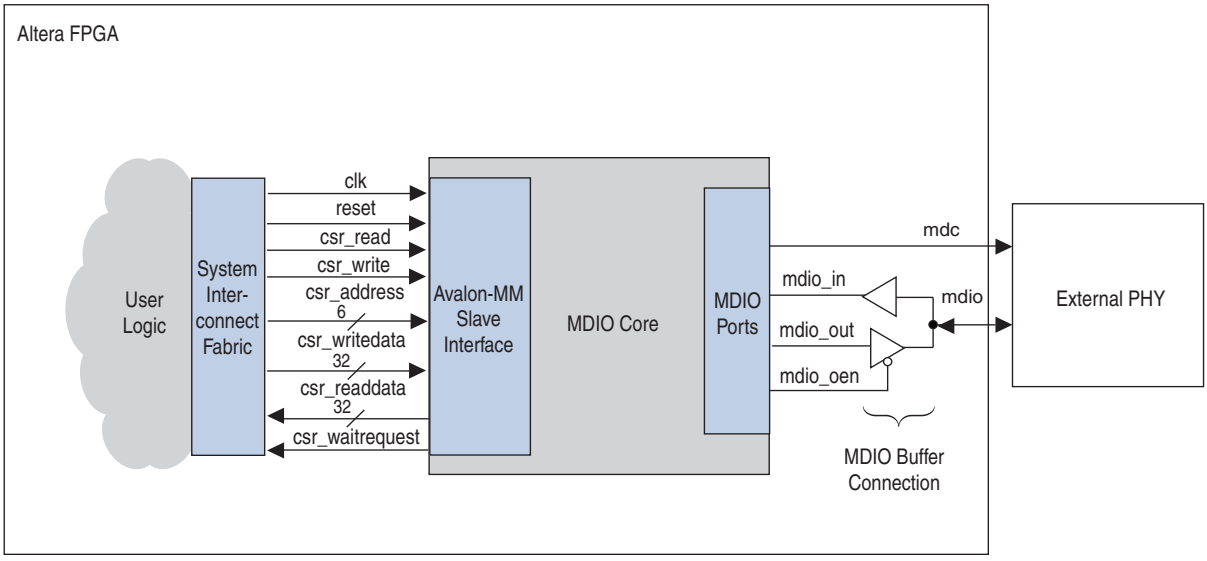
The core provides an Avalon Memory-Mapped (Avalon-MM) slave interface that allows Avalon-MM master peripherals (such as a CPU) to communicate with the core and access the external PHY by reading and writing the control and data registers. The system interconnect fabric connects the Avalon-MM master and slave interface.

A buffer connects the MDIO interface signals to the external PHY.

For more information about system interconnect fabric for Avalon-MM interfaces, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces*.

Figure 13–1 shows a block diagram of the MDIO core.

Figure 13–1. MDIO Core Block Diagram



MDIO Frame Format (Clause 45)

The MDIO core communicates with the external PHY device using frames. A complete frame is 64-bits long and consists of 32-bit preamble, 14-bit command, 2-bit bus direction change, and 16-bit data. Each bit is transferred on the rising edge of the MDIO's management data clock (MDC). The PHY management interface supports the standard MDIO specification (IEEE802.3 Ethernet Standard Clause 45).

Figure 13–2 illustrates the Clause 45 frame format.

Figure 13–2. MDIO Frame Format (Clause 45)

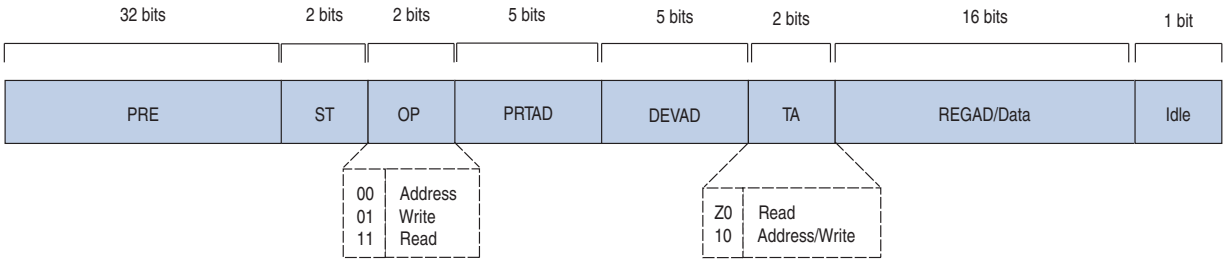


Table 13–1 describes the fields of the MDIO frame (Clause 45).

Table 13–1. MDIO Frame Field Descriptions—Clause 45 (Sheet 1 of 2)

Field Name	Description
PRE	Preamble. 32 bits of logical 1 sent prior to every transaction.
ST	The start of frame for indirect access cycles is indicated by the <00> pattern. This pattern assures a transition from the default one and identifies the frame as an indirect access.

Table 13–1. MDIO Frame Field Descriptions—Clause 45 (Sheet 2 of 2)

Field Name	Description
OP	The operation code field indicates the following transaction types: ■ 00 indicates that the frame payload contains the address of the register to access. ■ 01 indicates that the frame payload contains data to be written to the register whose address was provided in the previous address frame. ■ 11 indicates that the frame is a read operation. The post-read-increment-address operation <10> is not supported in this frame.
PRTAD	The port address (PRTAD) is 5 bits, allowing 32 unique port addresses. Transmission is MSB to LSB. A station management entity (STA) must have a prior knowledge of the appropriate port address for each port to which it is attached, whether connected to a single port or to multiple ports.
DEVAD	The device address (DEVAD) is 5 bits, allowing 32 unique MDIO manageable devices (MMDs) per port. Transmission is MSB to LSB.
TA	The turnaround time is a 2-bit time spacing between the device address field and the data field of a management frame to avoid contention during a read transaction. ■ For a read transaction, both the STA and the MMD remain in a high-impedance state (Z) for the first bit time of the turnaround. The MMD drives a 0 during the second bit time of the turnaround of a read or postread-increment-address transaction. ■ For a write or address transaction, the STA drives a 1 for the first bit time of the turnaround and a 0 for the second bit time of the turnaround.
REGAD/ Data	The register address (REGAD) or data field is 16 bits. For an address cycle, it contains the address of the register to be accessed on the next cycle. For the data cycle of a write frame, the field contains the data to be written to the register. For a read frame, the field contains the contents of the register. The first bit transmitted and received is bit 15.
Idle	The idle condition on MDIO is a high-impedance state. All tri-state drivers are disabled and the MMDs pullup resistor pulls the MDIO line to a one.

MDIO Clock Generation

The MDIO core's MDC is generated from the Avalon-MM interface clock signal, `clk`. The `MDC_DIVISOR` parameter specifies the division parameter. For more information about the parameter, refer to "Instantiating the Core in SOPC Builder".

The division factor must be defined such that the MDC frequency does not exceed 2.5 MHz.

Interfaces

The MDIO core consists of a single Avalon-MM slave interface. The slave interface performs Avalon-MM read and write transfers initiated by an Avalon-MM master in the client application logic. The Avalon-MM slave uses the `waitrequest` signal to implement backpressure on the Avalon-MM master for any read or write operation which has yet to be completed.

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

Operation

The MDIO core has bidirectional external signals to transfer data between the PHY and the core.

Write

To perform a write operation, you must first configure the device, port, and register addresses of the PHY by writing to the register at address `0x20`. Then, write to the `MDIO_ACCESS` register at address `0x21` to tell the core to generate an MDIO frame and write the data to the selected PHY device's register.

Read

To perform a read operation, first configure the device, port, and register addresses of the PHY by writing to the register at address `0x20`. Then, read the `MDIO_ACCESS` register at address `0x21` to read the selected PHY device's register.

Instantiating the Core in SOPC Builder

You can build a complete system in SOPC Builder. SOPC Builder automatically creates the system interconnect between components when you generate your system. The SOPC Builder uses the Avalon-MM bridge to connect the MDIO signals to the system interconnect fabric.

You can add the MDIO core to a system from the **System Contents** tab in SOPC Builder. In SOPC Builder, the MDIO core has the name **Ethernet MDIO**.

Table 13–2 lists and describes the parameter you can configure.

Table 13–2. Configurable Parameter

Parameter	Legal Values	Default Value	Description
MDC_DIVISOR	8-64	32	The host clock divisor provides the division factor for the clock on the Avalon-MM interface to generate the preferred MDIO clock (MDC). The division factor must be defined such that the MDC frequency does not exceed 2.5 MHz. Formula: $\frac{\text{Clock Source}}{\text{MDC_DIVISOR}} = \text{MDC Frequency}$ For example, if the Avalon-MM interface clock source is 100 MHz and the desired MDC frequency is 2.5 MHz, specify a value of 40 for the MDC_DIVISOR.

Register Map

An Avalon-MM master peripheral, such as a CPU, controls and communicates with the MDIO core via 32-bit registers, shown in Table 13–3.

Table 13–3. Register Map

Address Offset	Bit(s)	Name	Access Mode	Description
`0x00-0x1F`	[31:0]	Reserved	RW	Reserved for future use.
`0x20`	[4:0]	`MDIO_DEVAD`	RW	Contains the device address of the PHY.
	[7:5]	Reserved	RW	Unused.
	[12:8]	`MDIO_PRTAD`	RW	Contains the port address of the PHY.
	[15:13]	Reserved	RW	Unused.
	[31:16]	`MDIO_REGAD`	RW	Contains the register address of the PHY.
`0x21`	[31:0]	`MDIO_ACCESS`	RW	Performs a read or write of 32-bit data to the external PHY device. The addresses of the external PHY device's register, device, and port are specified in address offset `0x20`.

Device Support

The MDIO core supports the Stratix® IV GX and Stratix IV GT device families.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Volume 4: SOPC Builder* of the *Quartus II Handbook*

Document Revision History

Table 13–4 shows the revision history for this chapter.

Table 13–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	Initial release.	—

Section II. On-Chip Storage Peripherals

This section describes on-chip storage peripherals provided for SOPC Builder systems.

This section includes the following chapters:

- Chapter 14, Avalon-ST Single-Clock and Dual-Clock FIFO Cores
- Chapter 15, On-Chip FIFO Memory Core
- Chapter 16, Avalon-ST Multi-Channel Shared Memory FIFO Core

☞ For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

14. Avalon-ST Single-Clock and Dual-Clock FIFO Cores

Core Overview

The Avalon® Streaming (Avalon-ST) Single-Clock and Avalon-ST Dual-Clock FIFO cores are FIFO buffers which operate with a common clock and independent clocks for input and output ports respectively. The FIFO cores are configurable, SOPC Builder-ready, and integrate easily into any SOPC Builder-generated systems.

This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 14–4
- "Register Description" on page 14–5
- "Device Support" on page 14–6

Functional Description

Figure 14–1 and Figure 14–2 show block diagrams of the Avalon-ST Single-Clock FIFO and Avalon-ST Dual-Clock FIFO cores.

Figure 14–1. Avalon-ST Single Clock FIFO Core

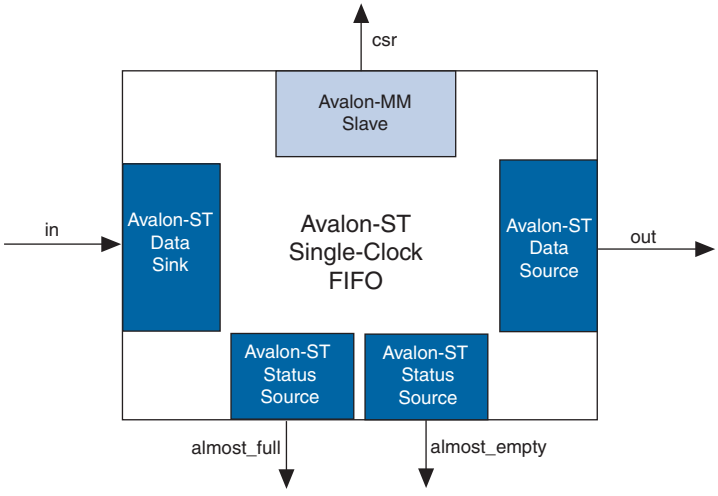
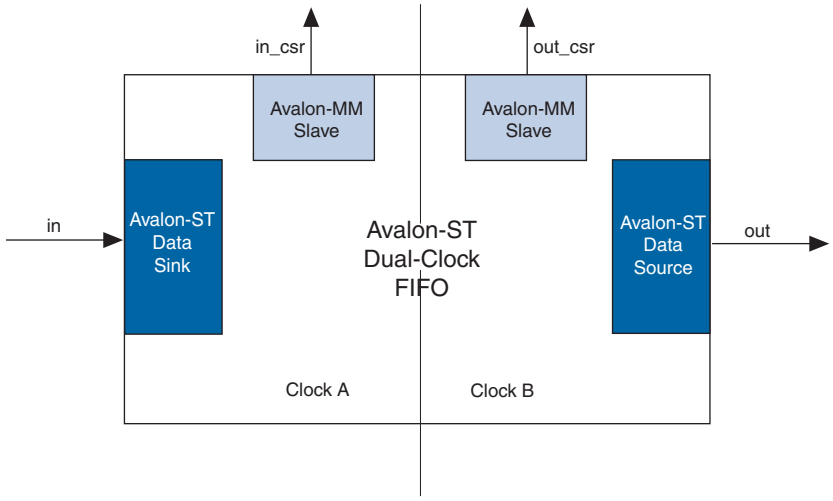


Figure 14–2. Avalon-ST Dual Clock FIFO Core



Interfaces

This section describes the interfaces implemented in the FIFO cores.

For more information about Avalon interfaces, refer to the *Avalon Interface Specifications*.

Avalon-ST Data Interface

Each FIFO core has an Avalon-ST data sink and source interfaces. The data sink and source interfaces in the dual-clock FIFO core are driven by different clocks.

Table 14–1 shows the properties of the Avalon-ST interfaces.

Table 14–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Configurable.
Channel	Supported, up to 255 channels.
Error	Configurable.
Packet	Configurable.

Avalon-MM Control and Status Register Interface

You can configure the single-clock FIFO core to include an optional Avalon-MM interface, and the dual-clock FIFO core to include an Avalon-MM interface in each clock domain. The Avalon-MM interface provides access to 32-bit registers, which allows you to retrieve the FIFO buffer fill level and configure the almost-empty and almost-full thresholds. In the single-clock FIFO core, you can also configure the packet and error handling modes. See Table 14–3 and Table 14–4 for the register descriptions.

Avalon-ST Status Interface

The single-clock FIFO core has two optional Avalon-ST status source interfaces from which you can obtain the FIFO buffer almost-full and almost empty statuses.

Operating Modes

The following lists the FIFO operating modes:

- Default mode—The core accepts incoming data on the `in` interface (Avalon-ST data sink) and forwards it to the `out` interface (Avalon-ST data source). The core asserts the `valid` signal on the Avalon-ST source interface to indicate that data is available at the interface.
- Store and forward mode—This mode only applies to the single-clock FIFO core. The core asserts the `valid` signal on the `out` interface only when a full packet of data is available at the interface.

  In this mode, you can also enable the drop-on-error feature by setting the `drop_on_error` register to 1. When this feature is enabled, the core drops all packets received with the `in_error` signal asserted.
- Cut-through mode— This mode only applies to the single-clock FIFO core. The core asserts the `valid` signal on the `out` interface to indicate that data is available for consumption when the number of entries specified in the `cut_through_threshold` register are available in the FIFO buffer.

To use the store and forward or cut-through mode, turn on the **Use store and forward** parameter to include the `csr` interface (Avalon-MM slave). Set the `cut_through_threshold` register to 0 to enable the store and forward mode; set the register to any value greater than 0 to enable the cut-through mode. The non-zero value specifies the minimum number of FIFO entries that must be available before the data is ready for consumption. Setting the register to 1 provides you with the default mode.

Fill Level

You can obtain the fill level of the FIFO buffer via the optional Avalon-MM control and status interface. Turn on the **Use fill level** parameter (**Use sink fill level** and **Use source fill level** in the dual-clock FIFO core) and read the `fill_level` register.

The dual-clock FIFO core has two fill levels, one in each clock domain. Due to the latency of the clock crossing logic, the fill levels reported in the input and output clock domains may be different at any given instance. In both cases, the fill level is pessimistic for the clock domain; the fill level is reported high in the input clock domain and low in the output clock domain.

The dual-clock FIFO has an output pipeline stage to improve f_{MAX}. This output stage is accounted for when calculating the output fill level, but not when calculating the input fill level. Hence, the best measure of the amount of data in the FIFO is given by the fill level in the output clock domain, while the fill level in the input clock domain represents the amount of space available in the FIFO (Available space = **FIFO depth** – input fill level).

Thresholds

You can use almost-full and almost-empty thresholds as a mechanism to prevent FIFO overflow and underflow. This feature is only available in the single-clock FIFO core.

To use the thresholds, turn on the **Use fill level**, **Use almost-full status**, and **Use almost-empty status** parameters. You can access the `almost_full_threshold` and `almost_full_threshold` registers via the csr interface and set the registers to an optimal value for your application. See Table 14–3 on page 14–5 for the register description.

You can obtain the almost-full and almost-empty statuses from `almost_full` and `almost_empty` interfaces (Avalon-ST status source). The core asserts the `almost_full` signal when the fill level is equal to or higher than the almost-full threshold. Likewise, the core asserts the `almost_empty` signal when the fill level is equal to or lower than the almost-empty threshold.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST Single Clock and Avalon-ST Dual Clock FIFO cores in SOPC Builder to add the cores to a system.

Table 14–2 lists and describes the parameters you can configure.

Table 14–2. Configurable Parameters (Part 1 of 2)

Parameter	Legal Values	Description
Bits per symbol	1–32	These parameters determine the width of the FIFO.
Symbols per beat	1–32	FIFO width = **Bits per symbol** * **Symbols per beat**, where: Bits per symbol is the number of bits in a symbol, and Symbols per beat is the number of symbols transferred in a beat.
Error width	0–32	The width of the `error` signal.
FIFO depth	1–32	The FIFO depth. An output pipeline stage is added to the FIFO to increase performance, which increases the FIFO depth by one.
Use packets	—	Turn on this parameter to enable packet support on the Avalon-ST data interfaces.
Channel width	1–32	The width of the `channel` signal.
Avalon-ST Single Clock FIFO Only		
Use fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface.
Avalon-ST Dual Clock FIFO Only		
Use sink fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface in the input clock domain.
Use source fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface in the output clock domain.
Write pointer synchronizer length	2–8	The length of the write pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability while increasing the latency of the core.
Read pointer synchronizer length	2–8	The length of the read pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability.

Table 14–2. Configurable Parameters (Part 2 of 2)

Parameter	Legal Values	Description
Use Max Channel	—	Turn on this parameter to specify the maximum channel number.
Max Channel	1–255	Maximum channel number.

For more information on metastability in Altera devices, refer to *AN 42: Metastability in Altera Devices*. For more information on metastability analysis and synchronization register chains, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Register Description

The `csr` interface in the Avalon-ST Single Clock FIFO core provides access to registers. Table 14–3 describes the registers.

Table 14–3. Register Description for Avalon-ST Single-Clock FIFO

32-Bit Word Offset	Name	Access	Reset	Description
0	`fill_level`	R	0	24-bit FIFO fill level. Bits 24 to 31 are unused.
1	Reserved	—	—	Reserved for future use.
2	`almost_full_threshold`	RW	**FIFO depth**–1	Set this register to a value that indicates the FIFO buffer is getting full.
3	`almost_empty_threshold`	RW	0	Set this register to a value that indicates the FIFO buffer is getting empty.
4	`cut_through_threshold`	RW	0	0—Enables store and forward mode. >0—Enables cut-through mode and specifies the minimum of entries in the FIFO buffer before the `valid` signal on the Avalon-ST source interface is asserted. Once the FIFO core starts sending the data to the downstream component, it continues to do so until the end of the packet. This register applies only when the **Use store and forward** parameter is turned on.
5	`drop_on_error`	RW	0	0—Disables drop-on error. 1—Enables drop-on error. This register applies only when the **Use packet** and **Use store and forward** parameters are turned on.

The `in_csr` and `out_csr` interfaces in the Avalon-ST Dual Clock FIFO core reports the FIFO fill level. Table 14–4 describes the fill level.

Table 14–4. Register Description for Avalon-ST Dual-Clock FIFO

32-Bit Word Offset	Name	Access	Reset Value	Description
0	`fill_level`	R	0	24-bit FIFO fill level. Bits 24 to 31 are unused.

Device Support

The Avalon-ST Single Clock and Avalon-ST Dual Clock FIFO cores support all Altera device families.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 14–5 shows the revision history for this chapter.

Table 14–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	■ Added description of the new features of the single-clock FIFO: store and forward mode, cut-through mode, and drop on error. ■ Added parameters and registers.	The changes reflect the enhancements made to the single-clock FIFO that include new features and new optional interfaces.
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	Added description of new parameters, **Write pointer synchronizer length** and **Read pointer synchronizer length**.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

15. On-Chip FIFO Memory Core

Core Overview

The on-chip FIFO memory core buffers data and provides flow control in an SOPC Builder system. The core can operate with a single clock or with separate clocks for the input and output ports, and it does not support burst read or write.

The input interface to the on-chip FIFO memory core may be an Avalon® Memory Mapped (Avalon-MM) write slave or an Avalon Streaming (Avalon-ST) sink. The output interface can be an Avalon-ST source or an Avalon-MM read slave. The data is delivered to the output interface in the same order that it was received at the input interface, regardless of the value of channel, packet, frame, or any other signals.

In single-clock mode, the on-chip FIFO memory core includes an optional status interface that provides information about the fill level of the FIFO core. In dual-clock mode, separate, optional status interfaces can be included for the input and output interfaces. The status interface also includes registers to set and control interrupts.

The on-chip FIFO memory core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. Device drivers are provided in the HAL system library allowing software to access the core using ANSI C.

This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 15–6
- "Instantiating the Core in SOPC Builder" on page 15–6
- "Software Programming Model" on page 15–8
- "Programming with the On-Chip FIFO Memory" on page 15–8
- "On-Chip FIFO Memory API" on page 15–13

Functional Description

The on-chip FIFO memory core has four configurations:

- Avalon-MM write slave to Avalon-MM read slave
- Avalon-ST sink to Avalon-ST source
- Avalon-MM write slave to Avalon-ST source
- Avalon-ST sink to Avalon-MM read slave

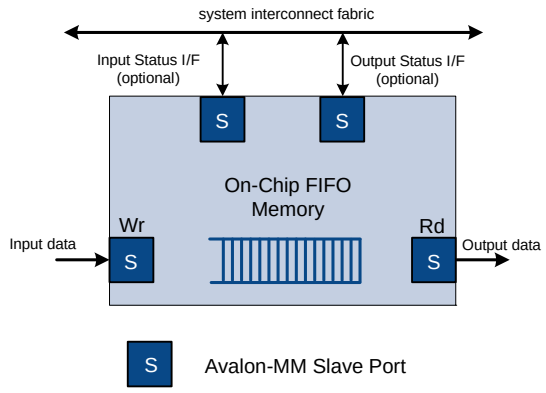
In all configurations, the input and output interfaces can use the optional backpressure signals to prevent underflow and overflow conditions. For the Avalon-MM interface, backpressure is implemented using the `waitrequest` signal. For Avalon-ST interfaces, backpressure is implemented using the `ready` and `valid` signals. For the on-chip FIFO memory core, the delay between the sink asserts `ready` and the source drives valid data is one cycle.

Avalon-MM Write Slave to Avalon-MM Read Slave

In this configuration, the input is a zero-address-width Avalon-MM write slave. An Avalon-MM write master pushes data into the FIFO core by writing to the input interface, and a read master (possibly the same master) pops data by reading from its output interface. The input and output data must be the same width.

If **Allow backpressure** is turned on, the `waitrequest` signal is asserted whenever the `data_in` master tries to write to a full FIFO buffer. `waitrequest` is only deasserted when there is enough space in the FIFO buffer for a new transaction to complete. `waitrequest` is asserted for read operations when there is no data to be read from the FIFO buffer, and is deasserted when the FIFO buffer has data.

Figure 15–1. FIFO with Avalon-MM Input and Output Interfaces

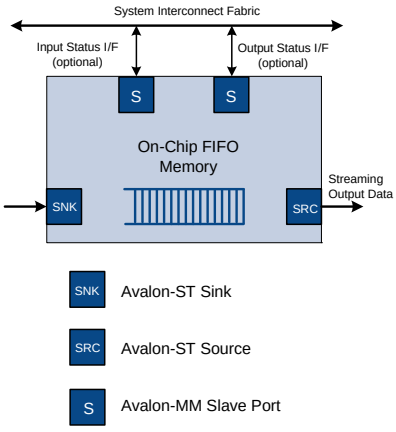


Avalon-ST Sink to Avalon-ST Source

This configuration has streaming input and output interfaces as illustrated in Figure 15–2. You can parameterize most aspects of the Avalon-ST interfaces including the **bits per symbol**, **symbols per beat**, and the width of `error` and `channel` signals. The input and output interfaces must be the same width. If **Allow backpressure** is turned on, both interfaces use the `ready` and `valid` signals to indicate when space is available in the FIFO core and when valid data is available.

For more information about the Avalon-ST interface protocol, refer to the *Avalon Interface Specifications*.

Figure 15–2. FIFO with Avalon-ST Input and Output Interfaces



Avalon-MM Write Slave to Avalon-ST Source

In this configuration, the input is an Avalon-MM write slave with a width of 32 bits as shown in Figure 15–3. The Avalon-ST output (source) data width must also be 32 bits. You can configure output interface parameters, including: **bits per symbol**, **symbols per beat**, and the width of the `channel` and `error` signals. The FIFO core performs the endian conversion to conform to the output interface protocol.

The signals that comprise the output interface are mapped into bits in the Avalon address space. If **Allow backpressure** is turned on, the input interface asserts `waitrequest` to indicate that the FIFO core does not have enough space for the transaction to complete.

Figure 15–3. FIFO with Avalon-MM Input Interface and Avalon-ST Output Interface

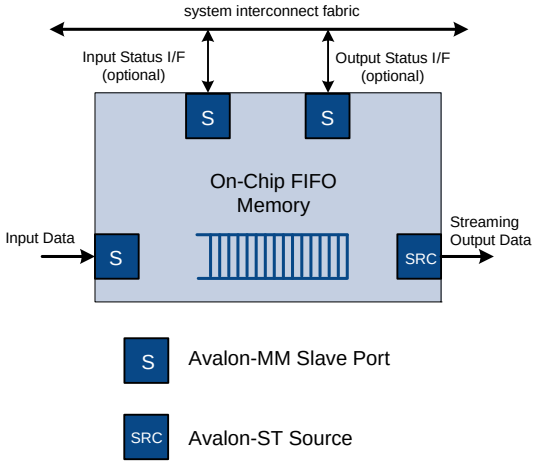


Table 15–1 shows the layout of the memory map for this configuration.

Table 15–1. Memory Map

Offset	Bits	Field	Description
0	31:0	SYMBOL_0, SYMBOL_1, SYMBOL_2 .. SYMBOL_n	Packet data. The value of the **Symbols per beat** parameter specifies the number of fields in this register; **Bits per symbol** specifies the width of each field.
1	0	SOP	The value of the `startofpacket` signal.
	1	EOP	The value of the `endofpacket` signal.
	6:2	EMPTY	The value of the `empty` signal.
	7	—	Reserved.
	15:8	CHANNEL	The value of the `channel` signal. The number of bits occupied corresponds to the width of the signal. For example, if the width of the channel signal is 5, bits 8 to 12 are occupied and bits 13 to 15 are unused.
	23:16	ERROR	The value of the `error` signal. The number of bits occupied corresponds to the width of the signal. For example, if the width of the error signal is 3, bits 16 to 18 are occupied and bits 19 to 23 are unused.
	31:24	—	Reserved.

If **Enable packet data** is turned off, the Avalon-MM write master writes all data at address offset 0 repeatedly to push data into the FIFO core.

If **Enable packet data** is turned on, the Avalon-MM write master starts by writing the `SOP`, `ERROR` (optional), `CHANNEL` (optional), `EOP`, and `EMPTY` packet status information at address offset 1. Writing to address offset 1 does not push data into the FIFO core. The Avalon-MM master then writes packet data to address offset 0 repeatedly, pushing 8-bit symbols into the FIFO core. Whenever a valid write occurs at address offset 0, the data and its respective packet information is pushed into the FIFO core. Subsequent data is written at address offset 0 without the need to clear the `SOP` field. Rewriting to address offset 1 is not required each time if the subsequent data to be pushed into the FIFO core is not the end-of-packet data, as long as `ERROR` and `CHANNEL` do not change.

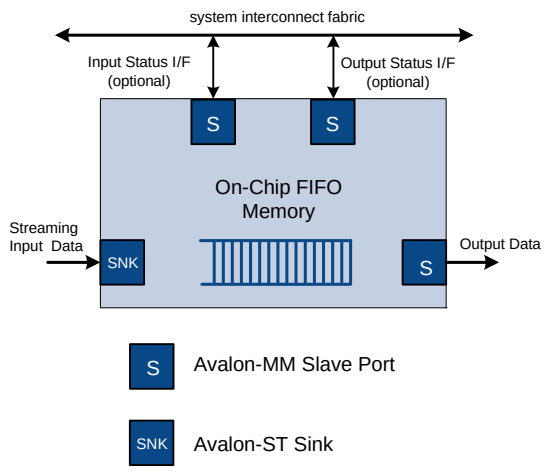
At the end of each packet, the Avalon-MM master writes to the address at offset 1 to set the `EOP` bit to 1, before writing the last symbol of the packet at offset 0. The write master uses the empty field to indicate the number of unused symbols at the end of the transfer. If the last packet data is not aligned with the **symbols per beat**, the `EMPTY` field indicates the number of empty symbols in the last packet data. For example, if the Avalon-ST interface has **symbols per beat** of 4, and the last packet only has 3 symbols, the empty field will be 1, indicating that one symbol (the least significant symbol in the memory map) is empty.

Avalon-ST Sink to Avalon-MM Read Slave

In this configuration, the input is an Avalon-ST sink and the output is an Avalon-MM read slave with a width of 32 bits (Figure 15–4). The Avalon-ST input (sink) data width must also be 32 bits. You can configure input interface parameters, including: **bits per symbol**, **symbols per beat**, and the width of the `channel` and `error` signals. The FIFO core performs the endian conversion to conform to the output interface protocol.

An Avalon-MM master reads the data from the FIFO core. The signals are mapped into bits in the Avalon address space. If **Allow backpressure** is turned on, the input (sink) interface uses the `ready` and `valid` signals to indicate when space is available in the FIFO core and when valid data is available. For the output interface, `waitrequest` is asserted for read operations when there is no data to be read from the FIFO core. It is deasserted when the FIFO core has data to send. The memory map for this configuration is exactly the same as for the Avalon-MM to Avalon-ST FIFO core. See Table 15–1 on page 15–4 for the memory map .

Figure 15–4. FIFO with Avalon-ST Input and Avalon-MM Output



If **Enable packet data** is turned off, read data repeatedly at address offset 0 to pop the data from the FIFO core.

If **Enable packet data** is turned on, the Avalon-MM read master starts reading from address offset 0. If the read is valid, that is, the FIFO core is not empty, both data and packet status information are popped from the FIFO core. The packet status information is obtained by reading at address offset 1. Reading from address offset 1 does not pop data from the FIFO core. The `ERROR`, `CHANNEL`, `SOP`, `EOP` and `EMPTY` fields are available at address offset 1 to determine the status of the packet data read from address offset 0.

The `EMPTY` field indicates the number of empty symbols in the data field. For example, if the Avalon-ST interface has symbols-per-beat of 4, and the last packet data only has 1 symbol, the `empty` field is 3 to indicate that 3 symbols (the 3 least significant symbols in the memory map) are empty.

Status Interface

The FIFO core provides two optional status interfaces, one for the master writing to the input interface and a second for the read master reading from the output interface. For FIFO cores that operate in a single domain, a single status interface is sufficient to monitor the status of the FIFO core. In the dual clocking scheme, a second status interface using the output clock is necessary to accurately monitor the status of the FIFO core in both clock domains.

Clocking Modes

When single-clock mode is used, the FIFO core being used is SCFIFO. When dual-clock mode is chosen, the FIFO core being used is DCFIFO. In dual-clock mode, input data and write-side status interfaces use the write side clock domain; the output data and read-side status interfaces use the read-side clock domain.

Device Support

The on-chip FIFO memory core supports all Altera® device families except the Hardcopy® series.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the on-chip FIFO memory core in SOPC Builder to specify the core configuration. The following sections describe the available options.

FIFO Settings

The following sections outline the settings that pertain to the FIFO core as a whole.

Depth

Depth indicates the depth of the FIFO buffer, in Avalon-ST beats or Avalon-MM words. The default depth is 16. When dual clock mode is used, the actual FIFO depth is equal to depth-3. This is due to clock crossing and to avoid FIFO overflow.

Clock Settings

The two options are **Single clock mode** and **Dual clock mode**. In **Single clock mode**, all interface ports use the same clock. In **Dual clock mode**, input data and input side status are on the input clock domain. Output data and output side status are on the output clock domain.

Status Port

The optional status ports are Avalon-MM slaves. To include the optional input side status interface, turn on **Create status interface for input** on the SOPC Builder MegaWizard. For FIFOs whose input and output ports operate in separate clock domains, you can include a second status interface by turning on **Create status interface for output**. Turning on **Enable IRQ for status ports** adds an interrupt signal to the status ports.

FIFO Implementation

This option determines if the FIFO core is built from registers or embedded memory blocks. The default is to construct the FIFO core from embedded memory blocks.

Interface Parameters

The following sections outline the options for the input and output interfaces.

Input

Available input interfaces are **Avalon-MM** write slave and **Avalon-ST** sink.

Output

Available output interfaces are **Avalon-MM** read slave and **Avalon-ST** source.

Allow Backpressure

When **Allow backpressure** is on, an Avalon-MM interface includes the `waitrequest` signal which is asserted to prevent a master from writing to a full FIFO buffer or reading from an empty FIFO buffer. An Avalon-ST interface includes the `ready` and `valid` signals to prevent underflow and overflow conditions.

Avalon-MM Port Settings

Valid **Data widths** are 8, 16, and 32 bits.

If Avalon-MM is selected for one interface and Avalon-ST for the other, the data width is fixed at 32 bits.

The Avalon-MM interface accesses data 4 bytes at a time. For data widths other than 32 bits, be careful of potential overflow and underflow conditions.

Avalon-ST Port Settings

The following parameters allow you to specify the size and error handling of the Avalon-ST port or ports:

- **Bits per symbol**
- **Symbols per beat**
- **Channel width**
- **Error width**

If the symbol size is not a power of two, it is rounded up to the next power of two. For example, if the **bits per symbol** is 10, the symbol will be mapped to a 16-bit memory location. With 10-bit symbols, the maximum number of **symbols per beat** is two.

Enable packet data provides an option for packet transmission.

Software Programming Model

The following sections describe the software programming model for the on-chip FIFO memory core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the on-chip FIFO memory core using its HAL API.

HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the on-chip FIFO memory via the familiar HAL API, rather than accessing the registers directly.

Software Files

Altera provides the following software files for the on-chip FIFO memory core:

- **altera_avalon_fifo_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_fifo_util.h**—This file defines functions to access the on-chip FIFO memory core hardware. It provides utilities to initialize the FIFO, read and write status, enable flags and read events.
- **altera_avalon_fifo.h**—This file provides the public interface to the on-chip FIFO memory
- **altera_avalon_fifo_util.c**—This file implements the utilities listed in **altera_avalon_fifo_util.h**.

Programming with the On-Chip FIFO Memory

This section describes the low-level software constructs for manipulating the on-chip FIFO memory core hardware. Table 15–2 lists all of the available functions.

Table 15–2. On-Chip FIFO Memory Functions (Part 1 of 2)

Function Name	Description
`altera_avalon_fifo_init()`	Initializes the FIFO.
`altera_avalon_fifo_read_status()`	Returns the integer value of the specified bit of the status register. To read all of the bits at once, use the `ALTERA_AVALON_FIFO_STATUS_ALL` mask.
`altera_avalon_fifo_read_ienable()`	Returns the value of the specified bit of the interrupt enable register. To read all of the bits at once, use the `ALTERA_AVALON_FIFO_EVENT_ALL` mask.
`altera_avalon_fifo_read_almostfull()`	Returns the value of the `almostfull` register.
`altera_avalon_fifo_read_almostempty()`	Returns the value of the `almostempty` register.
`altera_avalon_fifo_read_event()`	Returns the value of the specified bit of the event register. All of the event bits can be read at once by using the `ALTERA_AVALON_FIFO_STATUS_ALL` mask.
`altera_avalon_fifo_read_level()`	Returns the fill level of the FIFO.
`altera_avalon_fifo_clear_event()`	Clears the specified bits and the event register and performs error checking.

Table 15–2. On-Chip FIFO Memory Functions (Part 2 of 2)

Function Name	Description
`altera_avalon_fifo_write_ienable()`	Writes the specified bits of the `interruptenable` register and performs error checking.
`altera_avalon_fifo_write_almostfull()`	Writes the specified value to the `almostfull` register and performs error checking.
`altera_avalon_fifo_write_almostempty()`	Writes the specified value to the `almostempty` register and performs error checking.
`altera_avalon_fifo_write_fifo()`	Writes the specified data to the `write_address`.
`altera_avalon_fifo_write_other_info()`	Writes the packet status information to the `write_address`. Only valid when the **Enable packet data** option is turned on.
`altera_avalon_fifo_read_fifo()`	Reads data from the specified `read_address`.
`altera_avalon_fifo_read__other_info()`	Reads the packet status information from the specified `read_address`. Only valid when the **Enable packet data** option is turned on.

Software Control

Table 15–3 provides the register map for the `status` register. The layout of `status` register for the input and output interfaces is identical.

Table 15–3. FIFO Status Register Memory Map

offset	31 24	23 16	15 8	7	6	5	4	3	2	1	0
base	`fill_level`										
base + 1						`i_status`					
base + 2						`event`					
base + 3						`interrupt enable`					
base + 4	`almostfull`										
base + 5	`almostempty`										

Table 15–4 outlines the use of the various fields of the `status` register.

Table 15–4. FIFO Status Field Descriptions (Part 1 of 2)

Field	Type	Description
`fill_level`	RO	The instantaneous fill level of the FIFO, provided in units of symbols for a FIFO with an Avalon-ST FIFO and words for an Avalon-MM FIFO.
`i_status`	RO	A 6-bit register that shows the FIFO's instantaneous status. See Table 15–5 for the meaning of each bit field.
`event`	RW1C	A 6-bit register with exactly the same fields as `i_status`. When a bit in the `i_status` register is set, the same bit in the `event` register is set. The bit in the `event` register is only cleared when software writes a 1 to that bit.
`interruptenable`	RW	A 6-bit interrupt enable register with exactly the same fields as the `event` and `i_status` registers. When a bit in the `event` register transitions from a 0 to a 1, and the corresponding bit in `interruptenable` is set, the master Is interrupted.

Table 15–4. FIFO Status Field Descriptions (Part 2 of 2)

Field	Type	Description
`almostfull`	RW	A threshold level used for interrupts and status. Can be written by the Avalon-MM status master at any time. The default threshold value for DCFIFO is Depth-4. The default threshold value for SCFIFO is Depth-1. The valid range of the threshold value is from 1 to the default. 1 is used when attempting to write a value smaller than 1. The default is used when attempting to write a value larger than the default.
`almostempty`	RW	A threshold level used for interrupts and status. Can be written by the Avalon-MM status master at any time. The default threshold value for DCFIFO is 1. The default threshold value for SCFIFO is 1. The valid range of the threshold value is from 1 to the maximum allowable `almostfull` threshold. 1 is used when attempting to write a value smaller than 1. The maximum allowable is used when attempting to write a value larger than the maximum allowable.

Table 15–5 describes the instantaneous status bits.

Table 15–5. Status Bit Field Descriptions

Bit(s)	Name	Description
0	`FULL`	Has a value of 1 if the FIFO is currently full.
1	`EMPTY`	Has a value of 1 if the FIFO is currently empty.
2	`ALMOSTFULL`	Has a value of 1 if the fill level of the FIFO is greater than the `almostfull` value.
3	`ALMOSTEMPTY`	Has a value of 1 if the fill level of the FIFO is less than the `almostempty` value.
4	`OVERFLOW`	Is set to 1 for 1 cycle every time the FIFO overflows. The FIFO overflows when an Avalon write master writes to a full FIFO. OVERFLOW is only valid when **Allow backpressure** is off.
5	`UNDERFLOW`	Is set to 1 for 1 cycle every time the FIFO underflows. The FIFO underflows when an Avalon read master reads from an empty FIFO. UNDERFLOW is only valid when **Allow backpressure** is off.

Table 15–6 lists the bit fields of the `event` register. These fields are identical to those in the `status` register and are set at the same time; however, these fields are only cleared when software writes a one to clear (W1C). The `event` fields can be used to determine if a particular event has occurred.

Table 15–6. Event Bit Field Descriptions

Bit(s)	Name	Description
1	`E_FULL`	Has a value of 1 if the FIFO has been full and the bit has not been cleared by software.
0	`E_EMPTY`	Has a value of 1 if the FIFO has been empty and the bit has not been cleared by software.
3	`E_ALMOSTFULL`	Has a value of 1 if the fill level of the FIFO has been greater than the `almostfull` threshold value and the bit has not been cleared by software.
2	`E_ALMOSTEMPTY`	Has a value of 1 if the fill level of the FIFO has been less than the `almostempty` value and the bit has not been cleared by software.
4	`E_OVERFLOW`	Has a value of 1 if the FIFO has overflowed and the bit has not been cleared by software.
5	`E_UNDERFLOW`	Has a value of 1 if the FIFO has underflowed and the bit has not been cleared by software.

Table 15–7 provides a mask for the six STATUS fields. When a bit in the `event` register transitions from a zero to a one, and the corresponding bit in the `interruptenable` register is set, the master is interrupted.

Table 15–7. InterruptEnable Bit Field Descriptions

Bit(s)	Name	Description
1	`IE_FULL`	Enables an interrupt if the FIFO is currently full.
0	`IE_EMPTY`	Enables an interrupt if the FIFO is currently empty.
3	`IE_ALMOSTFULL`	Enables an interrupt if the fill level of the FIFO is greater than the value of the `almostfull` register.
2	`IE_ALMOSTEMPTY`	Enables an interrupt if the fill level of the FIFO is less than the value of the `almostempty` register.
4	`IE_OVERFLOW`	Enables an interrupt if the FIFO overflows. The FIFO overflows when an Avalon write master writes to a full FIFO.
5	`IE_UNDERFLOW`	Enables an interrupt if the FIFO underflows. The FIFO underflows when an Avalon read master reads from an empty FIFO.
6	`ALL`	Enables all 6 status conditions to interrupt.

Macros to access all of the registers are defined in **altera_avalon_fifo_regs.h.** For example, this file includes the following macros to access the `status` register.

```
#define ALTERA_AVALON_FIFO_LEVEL_REG          0
#define ALTERA_AVALON_FIFO_STATUS_REG         1
#define ALTERA_AVALON_FIFO_EVENT_REG          2
#define ALTERA_AVALON_FIFO_IENABLE_REG        3
#define ALTERA_AVALON_FIFO_ALMOSTFULL_REG     4
#define ALTERA_AVALON_FIFO_ALMOSTEMPTY_REG    5
```

For a complete list of predefined macros and utilities to access the on-chip FIFO hardware, see: **<install_dir>\quartus\sopc_builder\components\altera_avalon_fifo\HAL\inc\alatera_avalon_fifo.h** and **<install_dir>\quartus\sopc_builder\components\altera_avalon_fifo\HAL\inc\alatera_avalon_fifo_util.h.**

Software Example

Example 15–1 shows sample codes for the core.

Example 15–1. Sample Code for the On-Chip FIFO Memory (Part 1 of 2)

```
/*********************************************************************/
//Includes
#include "altera_avalon_fifo_regs.h"
#include "altera_avalon_fifo_util.h"
#include "system.h"
#include "sys/alt_irq.h"
#include <stdio.h>
#include <stdlib.h>

#define ALMOST_EMPTY 2
#define ALMOST_FULL OUTPUT_FIFO_OUT_FIFO_DEPTH-5

volatile int input_fifo_wrclk_irq_event;

void print_status(alt_u32 control_base_address)
{
   printf("--------------------------------------\n");
   printf("LEVEL = %u\n", altera_avalon_fifo_read_level(control_base_address) );
   printf("STATUS = %u\n", altera_avalon_fifo_read_status(control_base_address,
ALTERA_AVALON_FIFO_STATUS_ALL) );
   printf("EVENT = %u\n", altera_avalon_fifo_read_event(control_base_address,
ALTERA_AVALON_FIFO_EVENT_ALL) );
   printf("IENABLE = %u\n", altera_avalon_fifo_read_ienable(control_base_address,
ALTERA_AVALON_FIFO_IENABLE_ALL) );
   printf("ALMOSTEMPTY = %u\n",
altera_avalon_fifo_read_almostempty(control_base_address) );
   printf("ALMOSTFULL = %u\n\n",
altera_avalon_fifo_read_almostfull(control_base_address));
}

static void handle_input_fifo_wrclk_interrupts(void* context, alt_u32 id)
{
   /* Cast context to input_fifo_wrclk_irq_event's type. It is important
   * to declare this volatile to avoid unwanted compiler optimization.
   */
   volatile int* input_fifo_wrclk_irq_event_ptr = (volatile int*) context;

   /* Store the value in the FIFO's irq history register in *context. */
   *input_fifo_wrclk_irq_event_ptr =
altera_avalon_fifo_read_event(INPUT_FIFO_IN_CSR_BASE, ALTERA_AVALON_FIFO_EVENT_ALL);
   printf("Interrupt Occurs for %#x\n", INPUT_FIFO_IN_CSR_BASE);
   print_status(INPUT_FIFO_IN_CSR_BASE);

   /* Reset the FIFO's IRQ History register. */
   altera_avalon_fifo_clear_event(INPUT_FIFO_IN_CSR_BASE,
ALTERA_AVALON_FIFO_EVENT_ALL);
}

/* Initialize the fifo */
static int init_input_fifo_wrclk_control()
{
   int return_code = ALTERA_AVALON_FIFO_OK;

   /* Recast the IRQ History pointer to match the alt_irq_register() function
   * prototype. */
   void* input_fifo_wrclk_irq_event_ptr = (void*) &input_fifo_wrclk_irq_event;
   /* Enable all interrupts. */
```

Example 15–1. Sample Code for the On-Chip FIFO Memory (Part 2 of 2)

```
   /* Clear event register, set enable all irq, set almostempty and
   almostfull threshold */
   return_code = altera_avalon_fifo_init(INPUT_FIFO_IN_CSR_BASE,
                                         0, // Disabled interrupts
                                         ALMOST_EMPTY,
                                         ALMOST_FULL);
   /* Register the interrupt handler. */
   alt_irq_register( INPUT_FIFO_IN_CSR_IRQ,
input_fifo_wrclk_irq_event_ptr, handle_input_fifo_wrclk_interrupts );
   return return_code;
}
```

On-Chip FIFO Memory API

This section describes the application programming interface (API) for the on-chip FIFO memory core.

altera_avalon_fifo_init()

Prototype: `int altera_avalon_fifo_init(alt_u32 address, alt_u32 ienable, alt_u32 emptymark, alt_u32 fullmark)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`ienable`—the value to write to the `interruptenable` register

`emptymark`—the value for the almost empty threshold level

`fullmark`—the value for the almost full threshold level

Returns: Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_EVENT_CLEAR_ERROR` for clear errors, `ALTERA_AVALON_FIFO_IENABLE_WRITE_ERROR` for interrupt enable write errors, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` for errors writing the `almostfull` and `almostempty` registers.

Description: Clears the `event` register, writes the `interruptenable` register, and sets the `almostfull` register and `almostempty` registers.

altera_avalon_fifo_read_status()

Prototype: `int altera_avalon_fifo_read_status(alt_u32 address, alt_u32 mask)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`mask`—masks the read value from the `status` register

Returns: Returns the masked bits of the addressed register.

Description: Gets the addressed register bits—the `AND` of the value of the addressed register and the mask.

altera_avalon_fifo_read_ienable()

Prototype: `int altera_avalon_fifo_read_ienable(alt_u32 address, alt_u32 mask)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`mask`—masks the read value from the `interruptenable` register

Returns: Returns the logical `AND` of the `interruptenable` register and the mask.

Description: Gets the logical `AND` of the `interruptenable` register and the mask.

altera_avalon_fifo_read_almostfull()

Prototype: `int altera_avalon_fifo_read_almostfull(alt_u32 address)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

Returns: Returns the value of the `almostfull` register.

Description: Gets the value of the `almostfull` register.

altera_avalon_fifo_read_almostempty()

Prototype: `int altera_avalon_fifo_read_almostempty(alt_u32 address)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

Returns: Returns the value of the `almostempty` register.

Description: Gets the value of the `almostempty` register.

altera_avalon_fifo_read_event()

Prototype: `int altera_avalon_fifo_read_event(alt_u32 address, alt_u32 mask)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`mask`—masks the read value from the `event` register

Returns: Returns the logical `AND` of the `event` register and the mask.

Description: Gets the logical `AND` of the `event` register and the mask. To read single bits of the event register use the single bit masks, for example: `ALTERA_AVALON_FIFO_FIFO_EVENT_F_MSK`.
To read the entire `event` register use the full mask: `ALTERA_AVALON_FIFO_EVENT_ALL`.

altera_avalon_fifo_read_level()

Prototype:	`int altera_avalon_fifo_read_level(alt_u32 address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave
Returns:	Returns the fill level of the FIFO.
Description:	Gets the fill level of the FIFO.

altera_avalon_fifo_clear_event()

Prototype:	`int altera_avalon_fifo_clear_event(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave
	`mask`—the mask to use for bit-clearing (1 means clear this bit, 0 means do not clear)
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_EVENT_CLEAR_ERROR` if unsuccessful.
Description:	Clears the specified bits of the `event` register.

altera_avalon_fifo_write_ienable()

Prototype:	`int altera_avalon_fifo_write_ienable(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave
	`mask`—the value to write to the `interruptenable` register. See **altera_avalon_fifo_regs.h** for individual interrupt bit masks.
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_IENABLE_WRITE_ERROR` if unsuccessful.
Description:	Writes the specified bits of the `interruptenable` register.

altera_avalon_fifo_write_almostfull()

Prototype: `int altera_avalon_fifo_write_almostfull(alt_u32 address, alt_u32 data)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`data`—the value for the almost full threshold level

Returns: Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` if unsuccessful.

Description: Writes `data` to the `almostfull` register.

altera_avalon_fifo_write_almostempty()

Prototype: `int altera_avalon_fifo_write_almostempty(alt_u32 address, alt_u23 data)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`data`—the value for the almost empty threshold level

Returns: Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` if unsuccessful.

Description: Writes `data` to the `almostempty` register.

altera_avalon_write_fifo()

Prototype: `int altera_avalon_write_fifo(alt_u32 write_address, alt_u32 ctrl_address, alt_u32 data)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `write_address`—the base address of the FIFO write slave

`ctrl_address`—the base address of the FIFO control slave

`data`—the value to write to address offset 0 for Avalon-MM to Avalon-ST transfers, the value to write to the single address available for Avalon-MM to Avalon-MM transfers. See the *Avalon Interface Specifications* for the data ordering.

Returns: Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_FULL` if unsuccessful.

Description: Writes `data` to the specified address if the FIFO is not full.

altera_avalon_write_other_info()

Prototype: `int altera_avalon_write_other_info(alt_u32 write_address, alt_u32 ctrl_address, alt_u32 data)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `write_address`—the base address of the FIFO write slave

`ctrl_address`—the base address of the FIFO control slave

`data`—the packet status information to write to address offset 1 of the Avalon interface. See the *Avalon Interface Specifications* for the ordering of the packet status information.

Returns: Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_FULL` if unsuccessful.

Description: Writes the packet status information to the `write_address`. Only valid when **Enable packet data** is on.

altera_avalon_fifo_read_fifo()

Prototype: `int altera_avalon_fifo_read_fifo(alt_u32 read_address, alt_u32 ctrl_address)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `read_address`—the base address of the FIFO read slave

`ctrl_address`—the base address of the FIFO control slave

Returns: Returns the data from address offset 0, or 0 if the FIFO is empty.

Description: Gets the data addressed by `read_address`.

altera_avalon_fifo_read_other_info()

Prototype: `int altera_avalon_fifo_read_other_info(alt_u32 read_address)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `read_address`—the base address of the FIFO read slave

Returns: Returns the packet status information from address offset 1 of the Avalon interface. See the *Avalon Interface Specifications* for the ordering of the packet status information.

Description: Reads the packet status information from the specified `read_address`. Only valid when **Enable packet data** is on.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 15–8 shows the revision history for this chapter.

Table 15–8. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	Revised the description of the memory map.	—
November 2009 v9.1.0	Added description to the core overview.	The core does not support burst read or write.
March 2009 v9.0.0	Updated the description of the function altera_avalon_fifo_read_status().	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

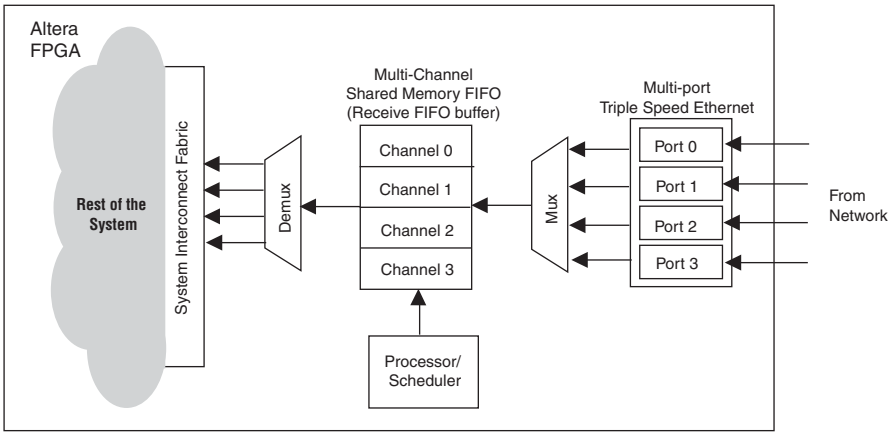
16. Avalon-ST Multi-Channel Shared Memory FIFO Core

Core Overview

The Avalon® Streaming (Avalon-ST) Multi-Channel Shared Memory FIFO core is a FIFO buffer with Avalon-ST data interfaces. The core, which supports up to 16 channels, is a contiguous memory space with dedicated segments of memory allocated for each channel. Data is delivered to the output interface in the same order it was received on the input interface for a given channel.

Figure 16–1 shows an example of how the core is used in a system. In this example, the core is used to buffer data going into and coming from a four-port Triple Speed Ethernet MegaCore function. A processor, if used, can request data for a particular channel to be delivered to the Triple Speed Ethernet MegaCore function.

Figure 16–1. Multi-Channel Shared Memory FIFO in a System—An Example



The Avalon-ST Multi-Channel Shared FIFO core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

This chapter contains the following sections:

- "Performance and Resource Utilization" on page 16–2
- "Functional Description" on page 16–3
- "Instantiating the Core in SOPC Builder" on page 16–5
- "Device Support" on page 16–5
- "Software Programming Model" on page 16–5

Performance and Resource Utilization

This section lists the resource utilization and performance data for various Altera device families. The estimates are obtained by compiling the core using the Quartus® II software.

Table 16–1 shows the resource utilization and performance data for a Stratix II GX device (EP2SGX130GF1508I4).

Table 16–1. Memory Utilization and Performance Data for Stratix II GX Devices

Channels	ALUTs	Logic Registers	Memory Blocks			f_{MAX} (MHz)
			M512	M4K	M-RAM	
4	559	382	0	0	1	> 125
12	1617	1028	0	0	6	> 125

Table 16–2 shows the resource utilization and performance data for a Stratix III device (EP3SL340F1760C3). The performance of the MegaCore function in Stratix IV devices is similar to Stratix III devices.

Table 16–2. Memory Utilization and Performance Data for Stratix III Devices

Channels	ALUTs	Logic Registers	Memory Blocks			f_{MAX} (MHz)
			M9K	M144K	MLAB	
4	557	345	37	0	0	> 125
12	1741	1028	0	24	0	> 125

Table 16–3 shows the resource utilization and performance data for a Cyclone III device (EP3C120F780I7).

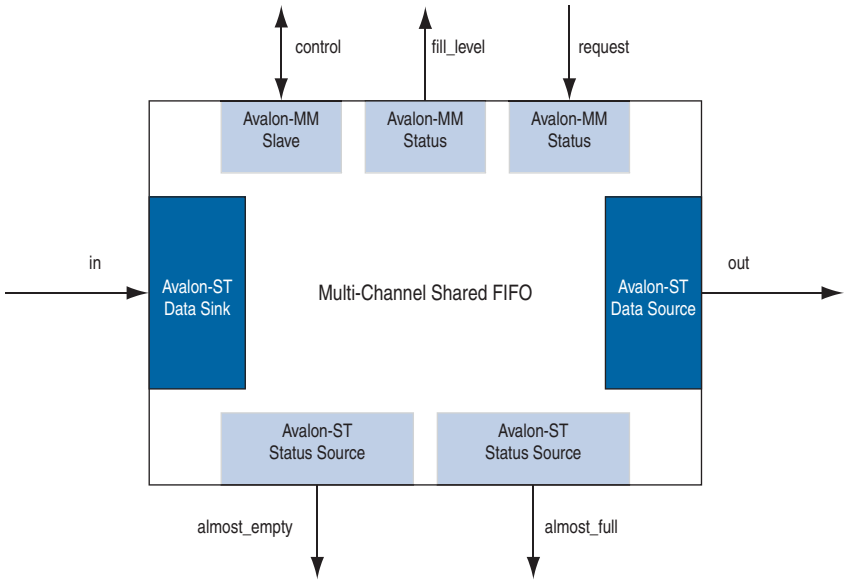
Table 16–3. Memory Utilization and Performance Data for Cyclone III Devices

Channels	Total Logic Elements	Total Registers	Memory M9K	f_{MAX} (MHz)
4	711	346	37	> 125
12	2284	1029	412	> 125

Functional Description

Figure 16–2 shows a block diagram of the Avalon-ST Multi-Channel Shared FIFO core.

Figure 16–2. Avalon-ST Multi-Channel Shared Memory FIFO Core



Interfaces

This section describes the core's interfaces.

Avalon-ST Interfaces

The core includes Avalon-ST interfaces for transferring data and almost-full status. Table 16–4 shows the properties of the Avalon-ST data interfaces.

Table 16–4. Properties of Avalon-ST Interfaces

Feature	Property	
	Data Interfaces	**Status Interfaces**
Backpressure	Ready latency = 0.	Not supported.
Data Width	Configurable.	Data width = 2 bits. Symbols per beat = 1.
Channel	Supported, up to 16 channels.	Supported, up to 16 channels.
Error	Configurable.	Not used.
Packet	Supported.	Not supported.

Avalon-MM Interfaces

The core can have up to three Avalon-MM interfaces:

- **Avalon-MM control interface**—Allows master peripherals to set and access almost-full and almost-empty thresholds. The same set of thresholds is used by all channels. See Table 16–6 for the description of the threshold registers.
- **Avalon-MM fill-level interface**—Allows master peripherals to retrieve the fill level of the FIFO buffer for a given channel. The fill level represents the amount of data in the FIFO buffer at any given time. The read latency on this interface is one. See Table 16–7 for the description of the fill-level registers.
- **Avalon-MM request interface**—Allows master peripherals to request data for a given channel. This interface is implemented only when the **Use Request** parameter is turned on. The `request_address` signal contains the channel number. Only one word of data is returned for each request.

For more information about Avalon interfaces, refer to the *Avalon Interface Specifications*.

Operation

The Avalon-ST Multi-Channel Shared FIFO core allocates dedicated memory segments within the core for each channel, and is implemented such that the memory segments occupy a single memory block. The parameter **FIFO depth** determines the depth of each memory segment.

The core receives data on its `in` interface (Avalon-ST sink) and stores the data in the allocated memory segments. If a packet contains any error (`in_error` signal is asserted), the core drops the packet.

When the core receives a request on its `request` interface (Avalon-MM slave), it forwards the requested data to its `out` interface (Avalon-ST source) only when it has received a full packet on its `in` interface. If the core has not received a full packet or has no data for the requested channel, it deasserts the `valid` signal on its `out` interface to indicate that data is not available for the channel. The output latency is three and only one word of data can be requested at a time.

When the Avalon-MM request interface is not in use, the `request_write` signal is kept asserted and the `request_address` signal is set to 0. Hence, if you configure the core to support more than one channel, you must also ensure that the **Use request** parameter is turned on. Otherwise, only channel 0 is accessible.

You can configure almost-full thresholds to manage FIFO overflow. The current threshold status for each channel is available from the core's Avalon-ST status interfaces in a round-robin fashion. For example, if the threshold status for channel 0 is available on the interface in clock cycle *n*, the threshold status for channel 1 is available in clock cycle *n+1* and so forth.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST Multi-Channel Shared FIFO core in SOPC Builder to add the core to a system.

Table 16–5 lists and describes the parameters you can configure.

Table 16–5. Configurable Parameters

Parameter	Legal Values	Description
Number of channels	1, 2, 4, 8, and 16	The total number of channels supported on the Avalon-ST data interfaces.
Symbols per beat	1–32	The number of symbols transferred in a beat on the Avalon-ST data interfaces
Bits per symbol	1–32	The symbol width in bits on the Avalon-ST data interfaces.
Error width	0–32	The width of the `error` signal on the Avalon-ST data interfaces.
FIFO depth	$2–2^{32}$	The depth of each memory segment allocated for a channel. The value must be a multiple of 2.
Address width	1–32	The width of the FIFO address. This parameter is determined by the parameter **FIFO depth**; **FIFO depth** = $2^{\text{Address Width}}$.
Use request	—	Turn on this parameter to implement the Avalon-MM request interface. If the core is configured to support more than one channel and the request interface is disabled, only channel 0 is accessible.
Use almost-full threshold 1	—	Turn on these parameters to implement the optional Avalon-ST almost-full and almost-empty interfaces and their corresponding registers. See Table 16–6 for the description of the threshold registers.
Use almost-full threshold 2	—	
Use almost-empty threshold 1	—	
Use almost-empty threshold 2	—	

Device Support

The Avalon-ST Multi-Channel Shared FIFO core supports all Altera device families.

Software Programming Model

The following sections describe the software programming model for the Avalon-ST Multi-Channel Shared FIFO core.

HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the Avalon-ST Multi-Channel Shared FIFO core via the familiar HAL API and the ANSI C standard library.

Register Map

You can configure the thresholds and retrieve the fill-level for each channel via the Avalon-MM control and fill-level interfaces respectively. Subsequent sections describe the registers accessible via each interface.

Control Register Interface

Table 16–6 shows the register map for the control interface.

Table 16–6. Control Interface Register Map

Byte Offset	Name	Access	Reset Value	Description
0	`ALMOST_FULL_THRESHOLD`	RW	0	Primary almost-full threshold. The bit `Almost_full_data[0]` on the Avalon-ST almost-full status interface is set to 1 when the FIFO level is equal to or greater than this threshold.
4	`ALMOST_EMPTY_THRESHOLD`	RW	0	Primary almost-empty threshold. The bit `Almost_empty_data[0]` on the Avalon-ST almost-empty status interface is set to 1 when the FIFO level is equal to or less than this threshold.
8	`ALMOST_FULL2_THRESHOLD`	RW	0	Secondary almost-full threshold. The bit `Almost_full_data[1]` on the Avalon-ST almost-full status interface is set to 1 when the FIFO level is equal to or greater than this threshold.
12	`ALMOST_EMPTY2_THRESHOLD`	RW	0	Secondary almost-empty threshold. The bit `Almost_empty_data[1]` on the Avalon-ST almost-empty status interface is set to 1 when the FIFO level is equal to or less than this threshold.

Fill-Level Register Interface

Table 16–7 shows the register map for the fill-level interface.

Table 16–7. Fill-level Interface Register Map

Byte Offset	Name	Access	Reset Value	Description
0	fill_level_0	RO	0	Fill level for each channel. Each register is defined for each channel. For example, if the core is configured to support four channel, four fill-level registers are defined.
4	fill_level_1	RO	0	
8	fill_level_2	RO	0	
(n*4)	fill_level_n	RO	0	

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 16–8 shows the revision history for this chapter.

Table 16–8. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	Added the description of almost-empty thresholds and fill-level registers. Revised the Operation section.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

Section III. Transport and Communication

This section describes communication and transport peripherals provided for SOPC Builder systems.

This section includes the following chapters:

- Chapter 17, SPI Slave/JTAG to Avalon Master Bridge Cores
- Chapter 18, Avalon Streaming Channel Multiplexer and Demultiplexer Cores
- Chapter 19, Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores
- Chapter 20, Avalon Packets to Transactions Converter Core
- Chapter 21, Avalon-ST Round Robin Scheduler Core
- Chapter 22, Avalon-ST Delay Core
- Chapter 23, Avalon-ST Splitter Core

☞ For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

17. SPI Slave/JTAG to Avalon Master Bridge Cores

Core Overview

The SPI Slave to Avalon® Master Bridge and the JTAG to Avalon Master Bridge cores provide a connection between host systems and SOPC Builder systems via the respective physical interfaces. Host systems can initiate Avalon Memory-Mapped (Avalon-MM) transactions by sending encoded streams of bytes via the cores' physical interfaces. The cores support reads and writes, but not burst transactions.

The SPI Slave to Avalon Master Bridge and the JTAG to Avalon Master Bridge are SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 17–3
- "Device Support" on page 17–3

Functional Description

Figure 17–1 shows a block diagram of the SPI Slave to Avalon Master Bridge core and its location in a typical system configuration.

Figure 17–1. SOPC Builder System with a SPI Slave to Avalon Master Bridge Core

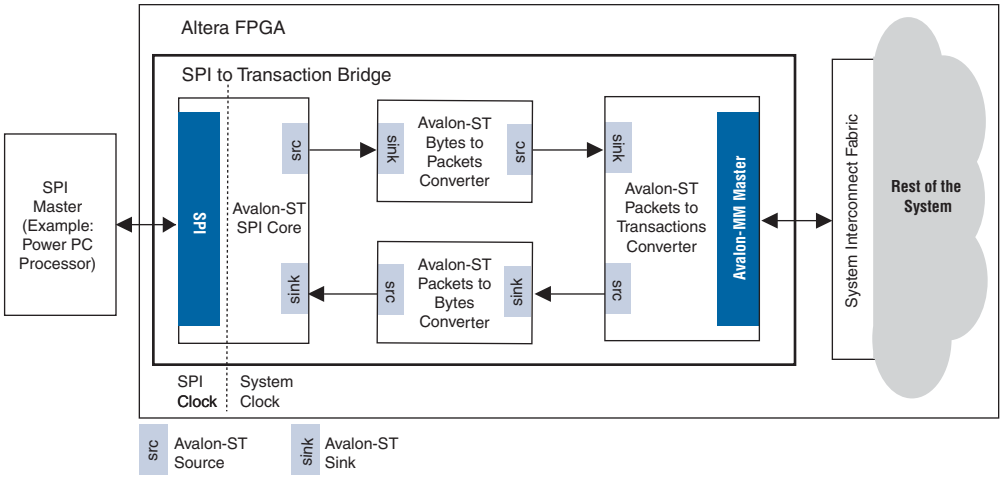
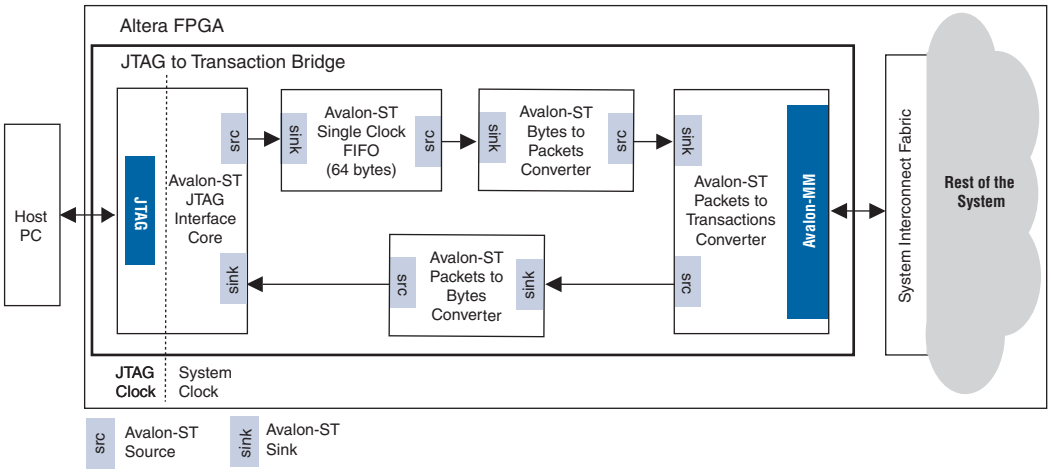


Figure 17–2 shows a block diagram of the JTAG to Avalon Master Bridge core and its location in a typical system configuration.

Figure 17–2. SOPC Builder System with a JTAG to Avalon Master Bridge Core



The SPI Slave to Avalon Master Bridge and the JTAG to Avalon Master Bridge cores accept encoded streams of bytes with transaction data on their respective physical interfaces and initiate Avalon-MM transactions on their Avalon-MM interfaces. Each bridge consists of the following cores, which are available as stand-alone components in SOPC Builder:

- **Avalon-ST Serial Peripheral Interface and Avalon-ST JTAG Interface**—Accepts incoming data in bits and packs them into bytes.
- **Avalon-ST Bytes to Packets Converter**—Transforms packets into encoded stream of bytes, and a likewise encoded stream of bytes into packets.
- **Avalon-ST Packets to Transactions Converter**—Transforms packets with data encoded according to a specific protocol into Avalon-MM transactions, and encodes the responses into packets using the same protocol.
- **Avalon-ST Single Clock FIFO**—Buffers data from the Avalon-ST JTAG Interface core. The FIFO is only used in the JTAG to Avalon Master Bridge.

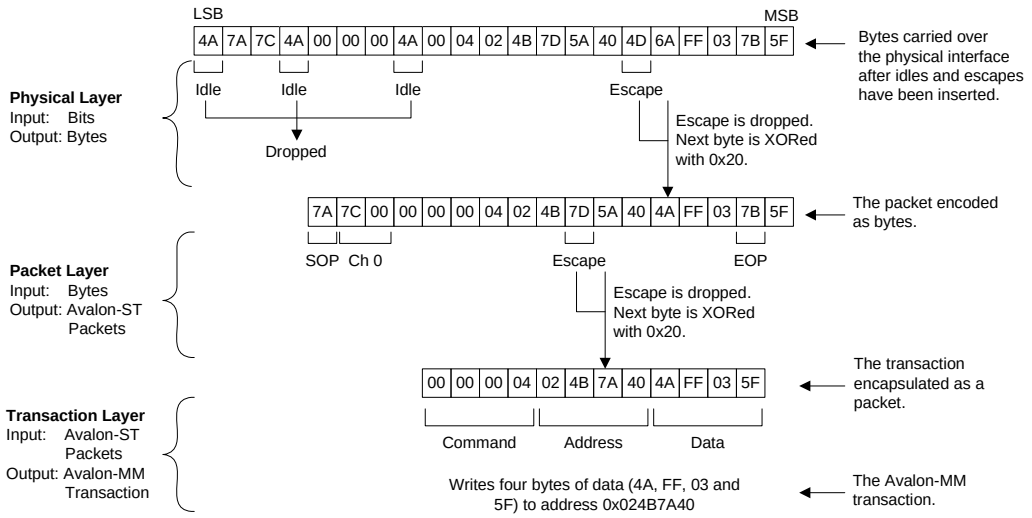
For the bridges to successfully transform the incoming streams of bytes to Avalon-MM transactions, the streams of bytes must be constructed according to the protocols used by the cores.

For more information about the protocol at each layer of the bridges and the single clock FIFO, refer to the following chapters:

- "Avalon-ST Serial Peripheral Interface Core" on page 10–1
- "Avalon-ST JTAG Interface Core" on page 31–1
- "Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores" on page 19–1
- "Avalon Packets to Transactions Converter Core" on page 20–1
- "Avalon-ST Single-Clock and Dual-Clock FIFO Cores" on page 14–1

The following example shows how a bytestream changes as it is transferred through the different layers in the bridges.

Figure 17–3. Bits to Avalon-MM Transaction



When the transaction is complete, the bridges send a response to the host system using the same protocol.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the SPI Slave to Avalon Master Bridge and the JTAG to Avalon Master Bridge in SOPC Builder to add the cores to a system. There are no user-configurable settings for the JTAG to Avalon Master Bridge core.

For the SPI Slave to Avalon Master Bridge core, the parameter **Number of synchronizer stages: Depth** allows you to specify the length of synchronization register chains. These register chains are used when a metastable event is likely to occur and the length specified determines the meantime before failure. The register chain length, however, affects the latency of the core.

For more information on metastability in Altera devices, refer to *AN 42: Metastability in Altera Devices*. For more information on metastability analysis and synchronization register chains, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Device Support

The SPI Slave to Avalon Master bridge supports all Altera® device families.

Referenced Documents

This chapter references the following documents:

- "Avalon-ST Serial Peripheral Interface Core" on page 10–1

- "Avalon-ST JTAG Interface Core" on page 31–1
- "Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores" on page 19–1
- "Avalon Packets to Transactions Converter Core" on page 20–1
- "Avalon-ST Single-Clock and Dual-Clock FIFO Cores" on page 14–1
- *AN 42: Metastability in Altera Devices*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 17–1 shows the revision history for this chapter.

Table 17–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	Added description of a new parameter **Number of synchronizer stages: Depth**.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

18. Avalon Streaming Channel Multiplexer and Demultiplexer Cores

Core Overview

The Avalon® streaming (Avalon-ST) channel multiplexer core receives data from a number of input interfaces and multiplexes the data into a single output interface, using the optional `channel` signal to indicate which input the output data is from. The Avalon-ST channel demultiplexer core receives data from a channelized input interface and drives that data to multiple output interfaces, where the output interface is selected by the input `channel` signal.

The multiplexer and demultiplexer can transfer data between interfaces on cores that support the unidirectional flow of data. The multiplexer and demultiplexer allow you to create multiplexed or de-multiplexer datapaths without having to write custom HDL code to perform these functions. The multiplexer includes a round-robin scheduler. Both cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Multiplexer" on page 18–2
- "Demultiplexer" on page 18–4
- "Device Support" on page 18–6
- "Hardware Simulation Considerations" on page 18–6
- "Software Programming Model" on page 18–6

Resource Usage and Performance

Resource utilization for the cores depends upon the number of input and output interfaces, the width of the datapath and whether the streaming data uses the optional packet protocol. For the multiplexer, the parameterization of the scheduler also effects resource utilization. Table 18–1 provides estimated resource utilization for eleven different configurations of the multiplexer.

Table 18–1. Multiplexer Estimated Resource Usage and Performance (Part 1 of 2)

No. of Inputs	Data Width	Scheduling Size (Cycles)	Stratix® II and Stratix II GX (Approximate LEs)		Cyclone® II		Stratix	
			f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	Y	1	500	31	420	63	422	80
2	Y	2	500	36	417	60	422	58
2	Y	32	451	43	364	68	360	49
8	Y	2	401	150	257	233	228	298
8	Y	32	356	151	219	207	211	123
16	Y	2	262	333	174	533	170	284
16	Y	32	310	337	161	471	157	277
2	N	1	500	23	400	48	422	52

Table 18–1. Multiplexer Estimated Resource Usage and Performance (Part 2 of 2)

No. of Inputs	Data Width	Scheduling Size (Cycles)	Stratix® II and Stratix II GX (Approximate LEs)		Cyclone® II		Stratix	
			f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	N	9	500	30	420	52	422	56
11	N	9	292	275	197	397	182	287
16	N	9	262	295	182	441	179	224

Table 18–2 provides estimated resource utilization for six different configurations of the demultiplexer. The core operating frequency varies with the device, the number of interfaces and the size of the datapath.

Table 18–2. Demultiplexer Estimated Resource Usage

No. of Inputs	Data Width (Symbols per Beat)	Stratix II (Approximate LEs)		Cyclone II		Stratix II GX (Approximate LEs)	
		f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	1	500	53	400	61	399	44
15	1	349	171	235	296	227	273
16	1	363	171	233	294	231	290
2	2	500	85	392	97	381	71
15	2	352	247	213	450	210	417
16	2	328	280	218	451	222	443

Multiplexer

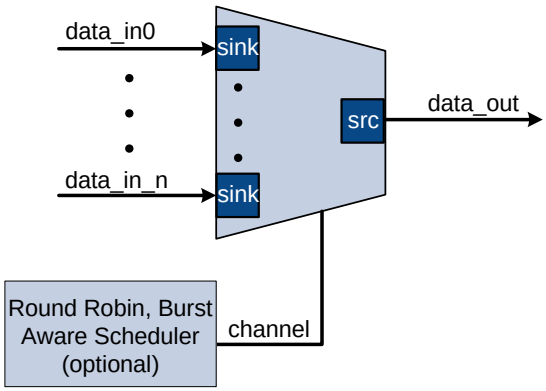
This section describes the hardware structure and functionality of the multiplexer component.

Functional Description

The Avalon-ST multiplexer takes data from a number of input data interfaces, and multiplexes the data onto a single output interface. The multiplexer includes a simple, round-robin scheduler that selects from the next input interface that has data. Each input interface has the same width as the output interface, so that all other input interfaces are backpressured when the multiplexer is carrying data from a different input interface.

The multiplexer includes an optional `channel` signal that enables each input interface to carry channelized data. When the `channel` signal is present on input interfaces, the multiplexer adds $\log_2$(num_input_interfaces) bits to make the output channel signal, such that the output channel signal has all of the bits of the input channel plus the bits required to indicate which input interface each cycle of data is from. These bits are appended to either the most or least significant bits of the output `channel` signal as specified in the SOPC Builder MegaWizard™ interface.

Figure 18–1. Multiplexer



The internal scheduler considers one input interface at a time, selecting it for transfer. Once an input interface has been selected, data from that input interface is sent until one of the following scenarios occurs:

- The specified number of cycles have elapsed.
- The input interface has no more data to send and `valid` is deasserted on a ready cycle.
- When packets are supported, `endofpacket` is asserted.

Input Interfaces

Each input interface is an Avalon-ST data interface that optionally supports packets. The input interfaces are identical; they have the same symbol and data widths, error widths, and channel widths.

Output Interface

The output interface carries the multiplexed data stream with data from all of the inputs. The symbol, data, and error widths are the same as the input interfaces. The width of the `channel` signal is the same as the input interfaces, with the addition of the bits needed to indicate the input each datum was from.

Instantiating the Multiplexer in SOPC Builder

Use the MegaWizard interface for the multiplexer core in SOPC Builder to specify the core configuration. The following sections list the available options in the MegaWizard interface.

Functional Parameters

You can configure the following options for the multiplexer:

- **Number of Input Ports**—The number of input interfaces that the multiplexer supports. Valid values are 2–16.
- **Scheduling Size (Cycles)**—The number of cycles that are sent from a single channel before changing to the next channel.

- **Use Packet Scheduling**—When this option is on, the multiplexer only switches the selected input interface on packet boundaries. Hence, packets on the output interface are not interleaved.
- **Use high bits to indicate source port**—When this option is on, the high bits of the output channel signal are used to indicate the input interface that the data came from. For example, if the input interfaces have 4-bit channel signals, and the multiplexer has 4 input interfaces, the output interface has a 6-bit channel signal. If this parameter is true, bits [5:4] of the output channel signal indicate the input interface the data is from, and bits [3:0] are the channel bits that were presented at the input interface.

Output Interface

You can configure the following options for the output interface:

- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1–32 bits.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1–32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**—The number of bits used for the `channel` signal for input interfaces. A value of 0 indicates that input interfaces do not have channels. A value of 4 indicates that up to 16 channels share the same input interface. The input channel can have a width between 0–31 bits. A value of 0 means that the optional `channel` signal is not used.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of 0 means the `error` signal is not used.

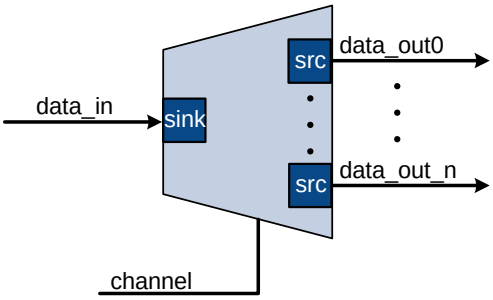
Demultiplexer

This section describes the hardware structure and functionality of the demultiplexer component.

Functional Description

That Avalon-ST demultiplexer takes data from a channelized input data interface and provides that data to multiple output interfaces, where the output interface selected for a particular transfer is specified by the input `channel` signal. The data is delivered to the output interfaces in the same order it was received at the input interface, regardless of the value of `channel`, `packet`, `frame`, or any other signal. Each of the output interfaces has the same width as the input interface, so each output interface is idle when the demultiplexer is driving data to a different output interface. The demultiplexer uses $\log_2$(num_output_interfaces) bits of the `channel` signal to select the output to which to forward the data; the remainder of the channel bits are forwarded to the appropriate output interface unchanged.

Figure 18–2. Demultiplexer



Input Interface

Each input interface is an Avalon-ST data interface that optionally supports packets.

Output Interfaces

Each output interface carries data from a subset of channels from the input interface. Each output interface is identical; all have the same symbol and data widths, error widths, and channel widths. The symbol, data, and error widths are the same as the input interface. The width of the `channel` signal is the same as the input interface, without the bits that were used to select the output interface.

Instantiating the Demultiplexer in SOPC Builder

Use the MegaWizard Interface for the demultiplexer core in SOPC Builder to specify the core configuration. The following sections list the available options in the MegaWizard Interface.

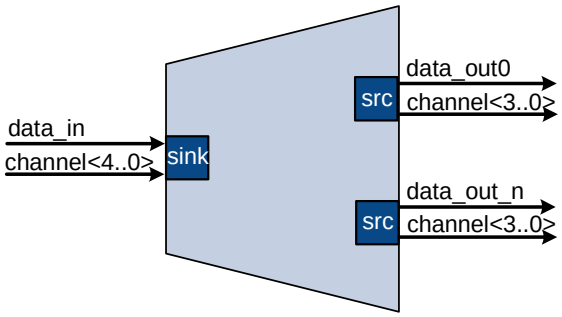
Functional Parameters

You can configure the following options for the demultiplexer as a whole:

- **Number of Output Ports**—The number of output interfaces that the multiplexer supports Valid values are 2–16.
- **High channel bits select output**—When this option is on, the high bits of the input channel signal are used by the de-multiplexing function and the low order bits are passed to the output. When this option is off, the low order bits are used and the high order bits are passed through.

  The following example illustrates the significance of the location of these signals. In Figure 18–3 there is one input interface and two output interfaces. If the low-order bits of the channel signal select the output interfaces, the even channels goes to channel 0 and the odd channels goes to channel 1. If the high-order bits of the channel signal select the output interface, channels 0–7 goes to channel 0 and channels 8–15 goes to channel 1.

Figure 18–3. Select Bits for Demultiplexer



Input Interface

You can configure the following options for the input interface:

- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1 to 32 bits.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**—The number of bits used for the `channel` signal for output interfaces. A value of 0 means that output interfaces do not use the optional `channel` signal.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of 0 means the `error` signal is not unused.

Device Support

The Avalon Streaming Channel Multiplexer and Demultiplexer cores support all Altera device families.

Hardware Simulation Considerations

The multiplexer and demultiplexer components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

The multiplexer and demultiplexer components do not have any user-visible control or status registers. Therefore, software cannot control or configure any aspect of the multiplexer or de-multiplexer at run-time. The components cannot generate interrupts.

Document Revision History

Table 18–3 shows the revision history for this chapter.

Table 18–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Added parameter **Include Packet Support**.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

19. Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores

Core Overview

The Avalon® Streaming (Avalon-ST) Bytes to Packets and Packets to Bytes Converter cores allow an arbitrary stream of packets to be carried over a byte interface, by encoding packet-related control signals such as `startofpacket` and `endofpacket` into byte sequences.The Avalon-ST Packets to Bytes Converter core encodes packet control and payload as a stream of bytes. The Avalon-ST Bytes to Packets Converter core accepts an encoded stream of bytes, and converts it into a stream of packets.

The SPI Slave to Avalon Master Bridge and JTAG to Avalon Master Bridge are examples of how the cores are used. For more information about the bridge, refer to "SPI Slave/JTAG to Avalon Master Bridge Cores" on page 17–1

Both of these cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 19–3
- "Device Support" on page 19–4

Functional Description

Figure 19–1 and Figure 19–2 show block diagrams of the Avalon-ST Bytes to Packets and Packets to Bytes Converter cores.

Figure 19–1. Avalon-ST Bytes to Packets Converter Core

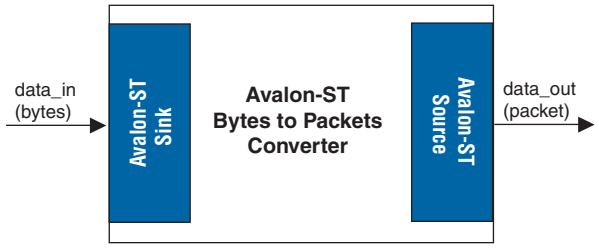
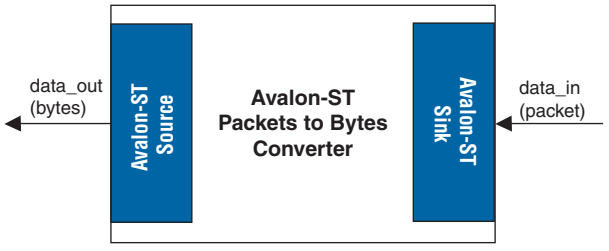


Figure 19–2. Avalon-ST Packets to Bytes Converter Core



Interfaces

Table 19–1 shows the properties of the Avalon-ST interfaces.

Table 19–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Supported, up to 255 channels.
Error	Not used.
Packet	Supported only on the Avalon-ST Bytes to Packet Converter core's source interface and the Avalon-ST Packet to Bytes Converter core's sink interface.

For more information about Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

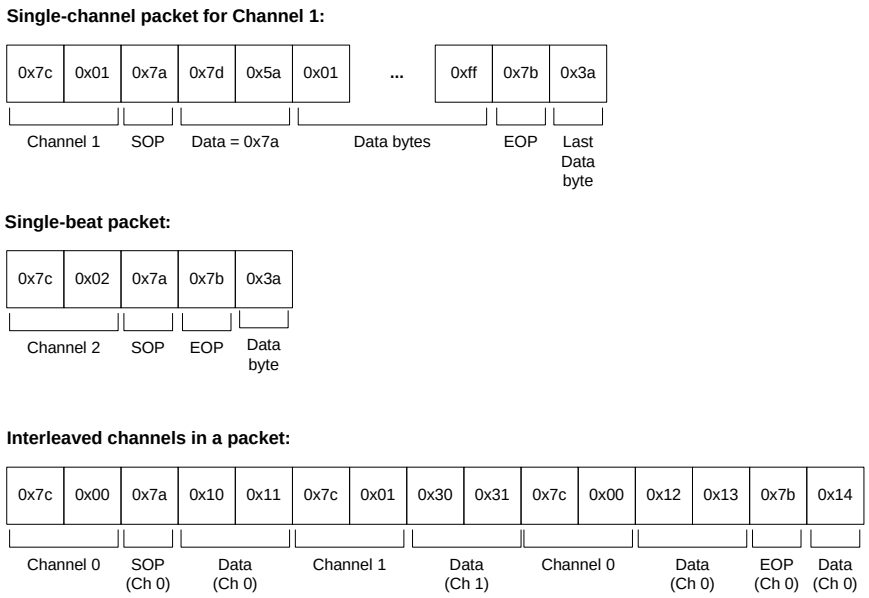
Operation—Avalon-ST Bytes to Packets Converter Core

The Avalon-ST Bytes to Packets Converter core receives streams of bytes and transforms them into packets. When parsing incoming bytestreams, the core decodes special characters in the following manner, with higher priority operations listed first:

- Escape (`0x7d`)—The core drops the byte. The next byte is XORed with `0x20`.
- Start of packet (`0x7a`)—The core drops the byte and marks the next payload byte as the start of a packet by asserting the `startofpacket` signal on the Avalon-ST source interface.
- End of packet (`0x7b`)—The core drops the byte and marks the following byte as the end of a packet by asserting the `endofpacket` signal on the Avalon-ST source interface. For single beat packets, both the `startofpacket` and `endofpacket` signals are asserted in the same clock cycle.
- Channel number indicator (`0x7c`)—The core drops the byte and takes the next non-special character as the channel number.

Figure 19–3 shows examples of bytestreams.

Figure 19–3. Examples of Bytestreams



Operation—Avalon-ST Packets to Bytes Converter Core

The Avalon-ST Packets to Bytes Converter core receives packetized data and transforms the packets to bytestreams. The core constructs outgoing bytestreams by inserting appropriate special characters in the following manner and sequence:

- If the `startofpacket` signal on the core's source interface is asserted, the core inserts the following special characters:
  - Channel number indicator (`0x7c`).
  - Channel number, escaping it if required.
  - Start of packet (`0x7a`).
- If the `endofpacket` signal on the core's source interface is asserted, the core inserts an end of packet (`0x7b`) before the last byte of data.
- If the `channel` signal on the core’s source interface changes to a new value within a packet, the core inserts a channel number indicator (`0x7c`) followed by the new channel number.
- If a data byte is a special character, the core inserts an escape (`0x7d`) followed by the data XORed with `0x20`.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST Bytes to Packets and Packets to Bytes Converter cores in SOPC Builder to add the core to a system. There are no user-configurable parameters for this core.

Device Support

The Avalon-ST Bytes to Packets and Packets to Bytes Converter cores support all Altera device families.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 19–2 shows the revision history for this chapter.

Table 19–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

20. Avalon Packets to Transactions Converter Core

Core Overview

The Avalon® Packets to Transactions Converter core receives streaming data from upstream components and initiates Avalon Memory-Mapped (Avalon-MM) transactions. The core then returns Avalon-MM transaction responses to the requesting components.

The SPI Slave to Avalon Master Bridge and JTAG to Avalon Master Bridge are examples of how this core is used. For more information on the bridge, refer to "SPI Slave/JTAG to Avalon Master Bridge Cores" on page 17–1

The Avalon Packets to Transactions Converter core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

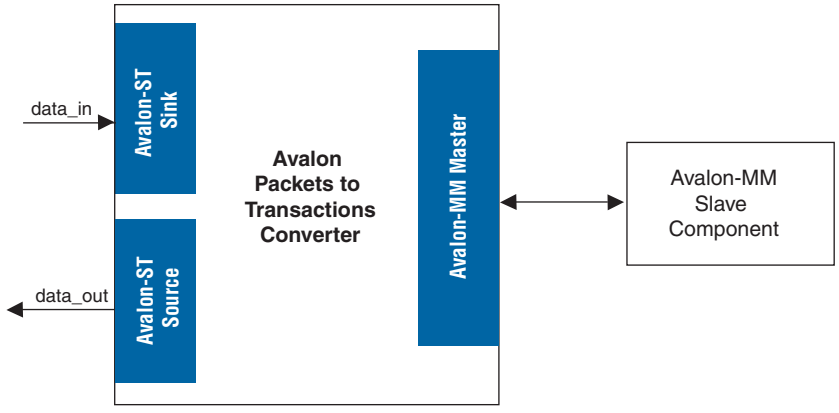
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 20–4
- "Device Support" on page 20–4

Functional Description

Figure 20–1 shows a block diagram of the Avalon Packets to Transactions Converter core.

Figure 20–1. Avalon Packets to Transactions Converter Core



Interfaces

Table 20–1 shows the properties of the Avalon-ST interfaces.

Table 20–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.
Error	Not used.
Packet	Supported.

The Avalon-MM master interface supports read and write transactions. The data width is set to 32 bits and burst transactions are not supported.

For more information about Avalon-ST interfaces, refer to *Avalon Interface Specifications*.

Operation

The Avalon Packets to Transactions Converter core receives streams of packets on its Avalon-ST sink interface and initiates Avalon-MM transactions. Upon receiving transaction responses from Avalon-MM slaves, the core transforms the responses to packets and returns them to the requesting components via its Avalon-ST source interface. The core does not report Avalon-ST errors.

Packet Formats

The core expects incoming data streams to be in the format shown in Table 20–2. A response packet is returned for every write transaction. The core also returns a response packet if a no transaction (0x7f) is received. An invalid transaction code is regarded as a no transaction. For read transactions, the core simply returns the data read.

Table 20–2. Packet Formats

Byte	Field	Description
Transaction Packet Format		
0	`Transaction code`	Type of transaction. See Table 20–1.
1	Reserved	Reserved for future use.
[3:2]	`Size`	Transaction size in bytes. For write transactions, the size indicates the size of the `data` field. For read transactions, the size indicates the total number of bytes to read.
[7:4]	`Address`	32-bit address for the transaction.
[n:8]	`Data`	Transaction data; data to be written for write transactions.
Response Packet Format		
0	`Transaction code`	The transaction code with the most significant bit inversed.
1	Reserved	Reserved for future use.
[4:2]	`Size`	Total number of bytes read/written successfully.

Supported Transactions

Table 20–3 lists the Avalon-MM transactions supported by the core.

Table 20–3. Transaction Supported

Transaction Code	Avalon-MM Transaction	Description
`0x00`	Write, non-incrementing address.	Writes data to the given address until the total number of bytes written to the same word address equals to the value specified in the `size` field.
`0x04`	Write, incrementing address.	Writes transaction data starting at the given address.
`0x10`	Read, non-incrementing address.	Reads 32 bits of data from the given address until the total number of bytes read from the same address equals to the value specified in the `size` field.
`0x14`	Read, incrementing address.	Reads the number of bytes specified in the `size` field starting from the given address.
`0x7f`	No transaction.	No transaction is initiated. You can use this transaction type for testing purposes. Although no transaction is initiated on the Avalon-MM interface, the core still returns a response packet for this transaction code.

The core can handle only a single transaction at a time. The `ready` signal on the core's Avalon-ST sink interface is asserted only when the current transaction is completely processed.

No internal buffer is implemented on the data paths. Data received on the Avalon-ST interface is forwarded directly to the Avalon-MM interface and vice-versa. Asserting the `waitrequest` signal on the Avalon-MM interface backpressures the Avalon-ST sink interface. In the opposite direction, if the Avalon-ST source interface is backpressured, the `read` signal on the Avalon-MM interface is not asserted until the backpressure is alleviated. Backpressuring the Avalon-ST source in the middle of a read could result in data loss. In such cases, the core returns the data that is successfully received.

A transaction is considered complete when the core receives an EOP. For write transactions, the actual data size is expected to be the same as the value of the `size` field. Whether or not both values agree, the core always uses the EOP to determine the end of data.

Malformed Packets

The following are examples of malformed packets:

- Consecutive start of packet (SOP)—An SOP marks the beginning of a transaction. If an SOP is received in the middle of a transaction, the core drops the current transaction without returning a response packet for the transaction, and initiates a new transaction. This effectively handles packets without an end of packet(EOP).
- Unsupported transaction codes—The core treats unsupported transactions as a no transaction.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon Packets to Transactions Converter core in SOPC Builder to add the core to a system. There are no user-configurable settings for this core.

Device Support

The Avalon Packets to Transactions Converter core supports all Altera device families.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 20–4 shows the revision history for this chapter.

Table 20–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

21. Avalon-ST Round Robin Scheduler Core

Core Overview

Avalon® Streaming (Avalon-ST) components in SOPC Builder provide a channel interface to stream data from multiple channels into a single component. In a multi-channel Avalon-ST component that stores data, the component can store data either in the sequence that it comes in (FIFO) or in segments according to the channel. When data is stored in segments according to channels, a scheduler is needed to schedule the read operations from that particular component. The most basic of the schedulers is the Avalon-ST Round Robin Scheduler core.

The Avalon-ST Round Robin Scheduler core is SOPC Builder-ready and can integrate easily into any SOPC Builder-generated systems.

This chapter contains the following sections:

- "Performance and Resource Utilization"
- "Functional Description" on page 21–2
- "Instantiating the Core in SOPC Builder" on page 21–4
- "Device Support" on page 21–4

Performance and Resource Utilization

This section lists the resource utilization and performance data for various Altera® device families. The estimates are obtained by compiling the core using the Quartus® II software.

Table 21–1 shows the resource utilization and performance data for a Stratix® II GX device (EP2SGX130GF1508I4).

Table 21–1. Resource Utilization and Performance Data for Stratix II GX Devices

Number of Channels	ALUTs	Logic Registers	Memory M512/M4K/ M-RAM	f_{MAX} (MHz)
4	7	7	0/0/0	> 125
12	25	17	0/0/0	> 125
24	62	30	0/0/0	> 125

Table 21–2 shows the resource utilization and performance data for a Stratix III device (EP3SL340F1760C3). The performance of the MegaCore® function in Stratix IV devices is similar to Stratix III devices.

Table 21–2. Resource Utilization and Performance Data for Stratix III Devices

Number of Channels	ALUTs	Logic Registers	Memory M9K/M144K/ MLAB	f_{MAX} (MHz)
4	7	7	0/0/0	> 125
12	25	17	0/0/0	> 125
24	67	30	0/0/0	> 125

Table 21–3 shows the resource utilization and performance data for a Cyclone® III device (EP3C120F780I7).

Table 21–3. Resource Utilization and Performance Data for Cyclone III Devices

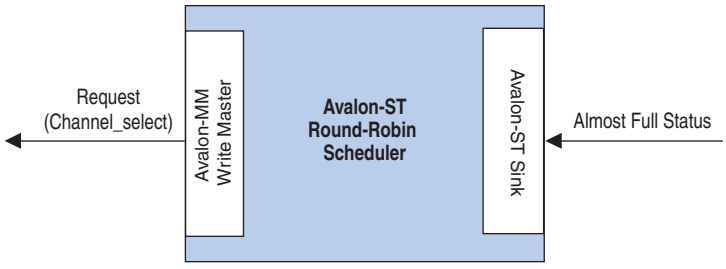
Number of Channels	Total Logic Elements	Total Registers	Memory M9K	f_{MAX} (MHz)
4	12	7	0	> 125
12	32	17	0	> 125
24	71	30	0	> 125

Functional Description

The Avalon-ST Round Robin Scheduler core controls the read operations from a multi-channel Avalon-ST component that buffers data by channels. It reads the almost-full threshold values from the multiple channels in the multi-channel component and issues the read request to the Avalon-ST source according to a round-robin scheduling algorithm.

Figure 21–1 shows the block diagram of the Avalon-ST Round Robin Scheduler.

Figure 21–1. Avalon-ST Round Robin Scheduler Block Diagram



Interfaces

The following interfaces are available in the Avalon-ST Round Robin Scheduler core:

- Almost-Full Status Interface
- Request Interface

Almost-Full Status Interface

The Almost-Full Status interface is an Avalon-ST sink interface. Table 21–4 describes the almost-full interface.

Table 21–4. Avalon-ST Interface Feature Support

Feature	Property
Backpressure	Not supported
Data Width	Data width = 1; Bits per symbol = 1
Channel	Maximum channel = 32; Channel width = 5
Error	Not supported
Packet	Not supported

The interface collects the almost-full status from the sink components for all the channels in the sequence provided.

Request Interface

The Request Interface is an Avalon Memory-Mapped (MM) Write Master interface. This interface requests data from a specific channel. The Avalon-ST Round Robin Scheduler core cycles through all of the channels it supports and schedules data to be read.

Operations

If a particular channel is almost full, the Avalon-ST Round Robin Scheduler will not schedule data to be read from that channel in the source component.

The Avalon-ST Round Robin Scheduler only requests 1 beat of data from a channel at each transaction. To request 1 beat of data from channel *n*, the scheduler writes the value `1` to address (4 ×*n*). For example, if the scheduler is requesting data from channel 3, the scheduler writes `1` to address `0xC`.

At every clock cycle, the Avalon-ST Round Robin Scheduler requests data from the next channel. Therefore, if the Avalon-ST Round Robin Scheduler starts requesting from channel 1, at the next clock cycle, it requests from channel 2. The Avalon-ST Round Robin Scheduler does not request data from a particular channel if the almost-full status for the channel is asserted. In this case, one clock cycle is used without a request transaction.

The Avalon-ST Round Robin Scheduler cannot determine if the requested component is able to service the request transaction. The component asserts `waitrequest` when it cannot accept new requests.

Table 21–5 shows the list of ports for the Avalon-ST Round Robin Scheduler core:

Table 21–5. Ports for the Avalon-ST Round Robin Scheduler (Part 1 of 2)

Signal	Direction	Description
Clock and Reset		
`clk`	In	Clock reference.
`reset_n`	In	Asynchronous active low reset.
Avalon-MM Request Interface		
`request_address` ($\log_2$ `Max_Channels-1:0)`	Out	The write address used to signal the channel the request is for.
`request_write`	Out	Write enable signal.

Table 21–5. Ports for the Avalon-ST Round Robin Scheduler (Part 2 of 2)

Signal	Direction	Description
`request_writedata`	Out	The amount of data requested from the particular channel. This value is always fixed at 1.
`request_waitrequest`	In	Wait request signal, used to pause the scheduler when the slave cannot accept a new request.
Avalon-ST Almost-Full Status Interface		
`almost_full_valid`	In	Indicates that `almost_full_channel` and `almost_full_data` are valid.
`almost_full_channel (Channel_Width-1:0)`	In	Indicates the channel for the current status indication.
`almost_full_data (log₂ Max_Channels-1:0)`	In	A 1-bit signal that is asserted high to indicate that the channel indicated by `almost_full_channel` is almost full.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST Round Robin Scheduler core in SOPC Builder to specify the core's configuration. Table 21–6 describes the parameters that can be configured for the Avalon-ST Round Robin Scheduler component.

Table 21–6. Parameters for Avalon-ST Round Robin Scheduler Component

Parameters	Values	Description
Number of channels	2–32	Specifies the number of channels the Avalon-ST Round Robin Scheduler supports.
Use almost-full status	0–1	Specifies whether the almost-full interface is used. If the interface is not used, the core always requests data from the next channel at the next clock cycle.

Device Support

The Avalon-ST Round Robin Scheduler core supports supports all Altera device families.

Document Revision History

Table 21–7 shows the revision history for this chapter.

Table 21–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—

Table 21–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

22. Avalon-ST Delay Core

Core Overview

The Avalon® Streaming (Avalon-ST) Delay core provides a solution to delay Avalon-ST transactions by a constant number of clock cycles. This core supports up to 16 clock cycle delays.

The Avalon-ST Delay core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

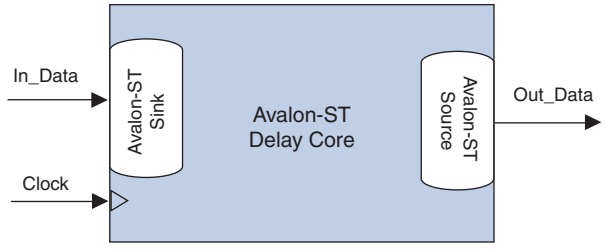
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 22–2
- "Device Support" on page 22–3

Functional Description

Figure 22–1 shows a block diagram of the Avalon-ST Delay core.

Figure 22–1. Avalon-ST Delay Core



The Avalon-ST Delay core adds a delay between the input and output interfaces. The core accepts all transactions presented on the input interface and reproduces them on the output interface N cycles later without changing the transaction.

The input interface delays the input signals by a constant (N) number of clock cycles to the corresponding output signals of the Avalon-ST output interface. The **Number Of Delay Clocks** parameter defines the constant (N) number, which must be between 0 and 16. The change of the `In_Valid` signal is reflected on the `Out_Valid` signal exactly N cycles later.

Reset

The Avalon-ST Delay core has a `reset` signal that is synchronous to the `clk` signal. When the core asserts the `reset` signal, the output signals are held at 0. After the `reset` signal is deasserted, the output signals are held at 0 for N clock cycles. The delayed values of the input signals are then reflected at the output signals after N clock cycles.

Interfaces

The Avalon-ST Delay core supports packetized and non-packetized interfaces with optional channel and error signals. This core does not support backpressure.

Table 22–1 shows the properties of the Avalon-ST interfaces.

Table 22–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Not supported.
Data Width	Configurable.
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

For more information about Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST Delay core in SOPC Builder to add the core to a system.

Table 22–2 lists and describes the parameters you can configure.

Table 22–2. Configurable Parameters

Parameter	Legal Values	Default Value	Description
Number Of Delay Clocks	0 to 16	1	Specifies the delay the core introduces, in clock cycles. The value of 0 is supported for some cases of parameterized systems in which no delay is required.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.
Use Packets	0 or 1	0	Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use Channel	0 or 1	0	The option to enable or disable the channel signal.
Channel Width	0-8	1	The width of the `channel` signal on the data interfaces. This parameter is disabled when **Use Channel** is set to 0.
Max Channels	0-255	1	The maximum number of channels that a data interface can support. This parameter is disabled when **Use Channel** is set to 0.
Use Error	0 or 1	0	The option to enable or disable the error signal.
Error Width	0–31	1	The width of the `error` signal on the output interfaces. A value of 0 indicates that the error signal is not in use. This parameter is disabled when **Use Error** is set to 0.

Device Support

The Avalon-ST Delay core supports all Altera® device families.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*

Document Revision History

Table 22–3 shows the revision history for this chapter.

Table 22–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
January 2010 v9.1.1	Initial release.	—

23. Avalon-ST Splitter Core

Core Overview

The Avalon® Streaming (Avalon-ST) Splitter core allows you to replicate transactions from an Avalon-ST source interface to multiple Avalon-ST sink interfaces. This core can support from 1 to 16 outputs.

The Avalon-ST Splitter core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

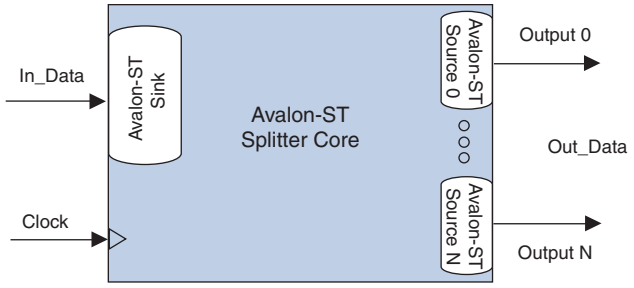
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 23–2
- "Device Support" on page 23–3

Functional Description

Figure 23–1 shows a block diagram of the Avalon-ST Splitter core.

Figure 23–1. Avalon-ST Splitter Core



The Avalon-ST Splitter core copies all input signals from the input interface to the corresponding output signals of each output interface without altering the size or functionality. This include all signals except for the `ready` signal.

The Avalon-ST Splitter core includes a clock signal used by SOPC Builder to determine the Avalon-ST interface and clock domain that this core resides in. Because the clock signal is unused internally, no latency is introduced when using this core.

Backpressure

The Avalon-ST Splitter core handles backpressure by AND-ing the `ready` signals from all of the output interfaces and sending the result to the input interface. This way, if any output interface deasserts the `ready` signal, the input interface receives the deasserted `ready` signal as well. This mechanism ensures that backpressure on any of the output interfaces is propagated to the input interface.

When the **Qualify Valid Out** parameter is set to 1, the `Out_Valid` signals on all other output interfaces are gated when backpressure is applied from one output interface. In this case, when any output interface deasserts its `ready` signal, the `Out_Valid` signals on the rest of the output interfaces are deasserted as well.

When the **Qualify Valid Out** parameter is set to 0, the output interfaces have a non-gated `Out_Valid` signal when backpressure is applied. In this case, when an output interface deasserts its ready signal, the `Out_Valid` signals on the rest of the output interfaces are not affected.

Because the logic is purely combinational, the core introduces no latency.

Interfaces

The Avalon-ST Splitter core supports packetized and non-packetized interfaces with optional channel and error signals. The core propagates backpressure from any output interface to the input interface.

Table 23–1 shows the properties of the Avalon-ST interfaces.

Table 23–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Configurable.
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

For more information about Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST Splitter core in SOPC Builder to add the core to a system.

Table 23–2 lists and describes the parameters you can configure.

Table 23–2. Configurable Parameters

Parameter	Legal Values	Default Value	Description
Number Of Outputs	1 to 16	2	The number of output interfaces. The value of 1 is supported for some cases of parameterized systems in which no duplicated output is required.
Qualify Valid Out	0 or 1	1	Determines whether the `Out_Valid` signal is gated or non-gated when backpressure is applied.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.

Table 23–2. Configurable Parameters

Parameter	Legal Values	Default Value	Description
Use Packets	0 or 1	0	Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use Channel	0 or 1	0	The option to enable or disable the channel signal.
Channel Width	0-8	1	The width of the `channel` signal on the data interfaces. This parameter is disabled when **Use Channel** is set to 0.
Max Channels	0-255	1	The maximum number of channels that a data interface can support. This parameter is disabled when **Use Channel** is set to 0.
Use Error	0 or 1	0	The option to enable or disable the error signal.
Error Width	0–31	1	The width of the `error` signal on the output interfaces. A value of 0 indicates that the error signal is not used. This parameter is disabled when **Use Error** is set to 0.

Device Support

The Avalon-ST Splitter core supports all Altera® device families.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*

Document Revision History

Table 23–3 shows the revision history for this chapter.

Table 23–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
January 2010 v9.1.1	Initial release.	—

Section IV. Peripherals

This section describes multiprocessor coordination peripherals provided by Altera® for SOPC Builder systems. These components provide reliable mechanisms for multiple Nios® II processors to communicate with each other, and coordinate operations.

This section includes the following chapters:

- Chapter 24, Scatter-Gather DMA Controller Core
- Chapter 25, DMA Controller Core
- Chapter 26, Video Sync Generator and Pixel Converter Cores
- Chapter 27, Interval Timer Core
- Chapter 28, Mutex Core
- Chapter 29, Mailbox Core
- Chapter 30, Vectored Interrupt Controller Core

☞ For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

24. Scatter-Gather DMA Controller Core

Core Overview

The Scatter-Gather Direct Memory Access (SG-DMA) controller core implements high-speed data transfer between two components. You can use the SG-DMA controller core to transfer data from:

- Memory to memory
- Data stream to memory
- Memory to data stream

The SG-DMA controller core transfers and merges non-contiguous memory to a continuous address space, and vice versa. The core reads a series of descriptors that specify the data to be transferred.

For applications requiring more than one DMA channel, multiple instantiations of the core can provide the required throughput. Each SG-DMA controller has its own series of descriptors specifying the data transfers. A single software module controls all of the DMA channels.

The SG-DMA controller core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios® II processor, device drivers are provided in the Hardware Abstraction Layer (HAL) system library, allowing software to access the core using the provided driver.

Example Systems

Figure 24–1 shows a SG-DMA controller core in a block diagram for the DMA subsystem of a printed circuit board. The SG-DMA core in the FPGA reads streaming data from an internal streaming component and writes data to an external memory. A Nios II processor provides overall system control.

Figure 24–1. SG-DMA Controller Core with Streaming Peripheral and External Memory

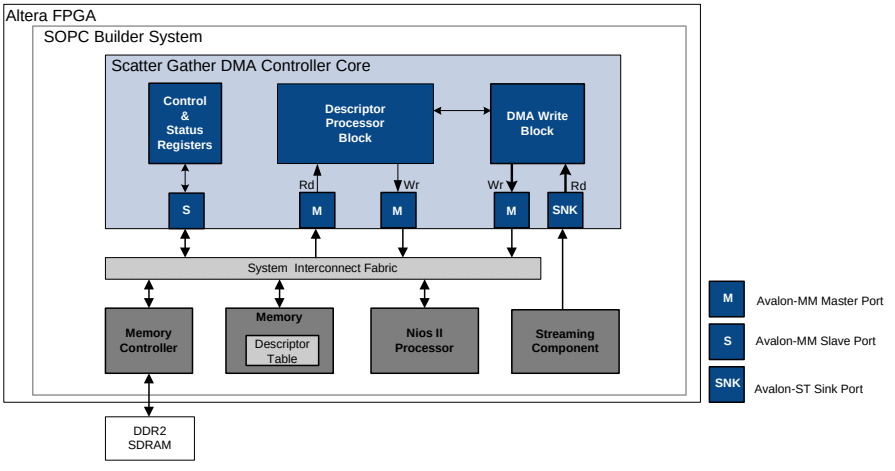
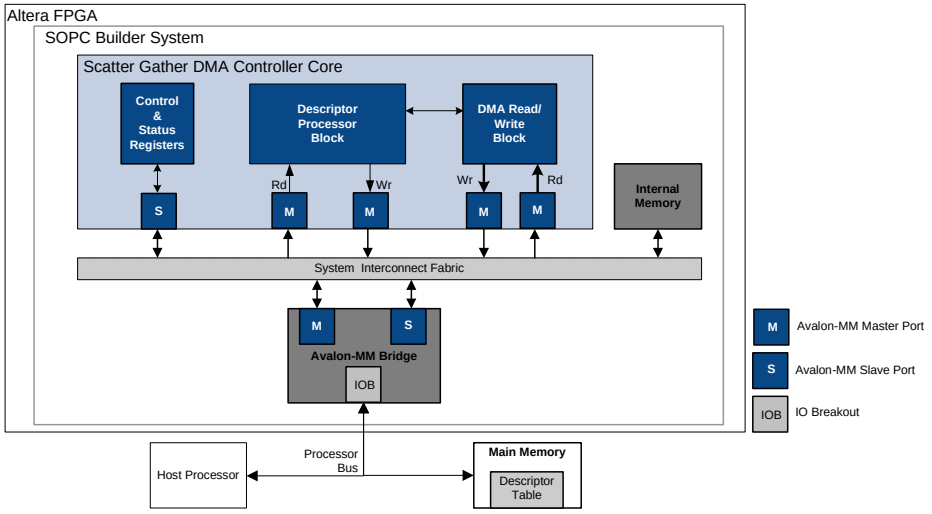


Figure 24–2 shows a different use of the SG-DMA controller core, where the core transfers data between an internal and external memory. The host processor and memory are connected to a system bus, typically either a PCI Express or Serial RapidIO™.

Figure 24–2. SG-DMA Controller Core with Internal and External Memory



Comparison of SG-DMA Controller Core and DMA Controller Core

The SG-DMA controller core provides a significant performance enhancement over the previously available DMA controller core, which could only queue one transfer at a time. Using the DMA Controller core, a CPU had to wait for the transfer to complete before writing a new descriptor to the DMA slave port. Transfers to non-contiguous memory could not be linked; consequently, the CPU overhead was substantial for small transfers, degrading overall system performance. In contrast, the SG-DMA controller core reads a series of descriptors from memory that describe the required transactions and performs all of the transfers without additional intervention from the CPU.

In This Chapter

This chapter contains the following sections:

- "Functional Description" on page 24–3
- "Device Support" on page 24–9
- "Instantiating the Core in SOPC Builder" on page 24–9
- "Simulation Considerations" on page 24–10
- "Software Programming Model" on page 24–10
- "Programming with SG-DMA Controller" on page 24–15

Resource Usage and Performance

Resource utilization for the core is 600–1400 logic elements, depending upon the width of the datapath, the parameterization of the core, the device family, and the type of data transfer. Table 24–1 provides the estimated resource usage for a SG-DMA controller core used for memory to memory transfer. The core is configurable and the resource utilization varies with the configuration specified.

Table 24–1. SG-DMA Estimated Resource Usage

Datapath	Cyclone® II	Stratix® (LEs)	Stratix II (ALUTs)
8-bit datapath	850	650	600
32-bit datapath	1100	850	700
64-bit datapath	1250	1250	800

The core operating frequency varies with the device and the size of the datapath. Table 24–2 provides an example of expected performance for SG-DMA cores instantiated in several different device families.

Table 24–2. SG-DMA Peak Performance

Device	Datapath	f_{MAX}	Throughput
Cyclone II	64 bits	150 MHz	9.6 Gbps
Cyclone III	64 bits	160 MHz	10.2 Gbps
Stratix II/Stratix II GX	64 bits	250 MHz	16.0 Gbps
Stratix III	64 bits	300 MHz	19.2 Gbps

Functional Description

The SG-DMA controller core comprises three major blocks: descriptor processor, DMA read, and DMA write. These blocks are combined to create three different configurations:

- Memory to memory
- Memory to stream
- Stream to memory

The type of devices you are transferring data to and from determines the configuration to implement. Examples of memory-mapped devices are PCI, PCIe and most memory devices. The Triple Speed Ethernet MAC, DSP MegaCore functions and many video IPs are examples of streaming devices. A recompilation is necessary each time you change the configuration of the SG-DMA controller core.

Functional Blocks and Configurations

The following sections describe each functional block and configuration.

Descriptor Processor

The descriptor processor reads descriptors from the descriptor list via its Avalon® Memory-Mapped (MM) read master port and pushes commands into the command FIFOs of the DMA read and write blocks. Each command includes the following fields to specify a transfer:

- Source address
- Destination address
- Number of bytes to transfer
- Increment read address after each transfer
- Increment write address after each transfer
- Generate start of packet (SOP) and end of packet (EOP)

After each command is processed by the DMA read or write block, a *status token* containing information about the transfer such as the number of bytes actually written is returned to the descriptor processor, where it is written to the respective fields in the descriptor.

DMA Read Block

The DMA read block is used in memory-to-memory and memory-to-stream configurations. The block performs the following operations:

- Reads commands from the input command FIFO.
- Reads a block of memory via the Avalon-MM read master port for each command.
- Pushes data into the data FIFO.

If burst transfer is enabled, an internal read FIFO with a depth of twice the maximum read burst size is instantiated. The DMA read block initiates burst reads only when the read FIFO has sufficient space to buffer the complete burst.

DMA Write Block

The DMA write block is used in memory-to-memory and stream-to-memory configurations. The block reads commands from its input command FIFO. For each command, the DMA write block reads data from its Avalon-ST sink port and writes it to the Avalon-MM master port.

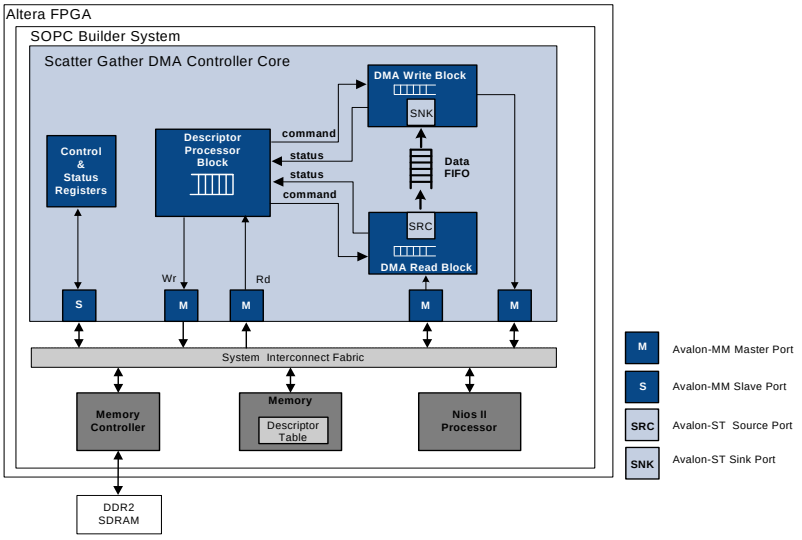
If burst transfer is enabled, an internal write FIFO with a depth of twice the maximum write burst size is instantiated. Each burst write transfers a fixed amount of data equals to the write burst size, except for the last burst. In the last burst, the remaining data is transferred even if the amount of data is less than the write burst size.

Memory-to-Memory Configuration

Memory-to-memory configurations include all three blocks: descriptor processor, DMA read, and DMA write. An internal FIFO is also included to provide buffering and flow control for data transferred between the DMA read and write blocks.

Figure 24–3 illustrates one possible memory-to-memory configuration with an internal Nios II processor and descriptor list.

Figure 24–3. Example of Memory-to-Memory Configuration

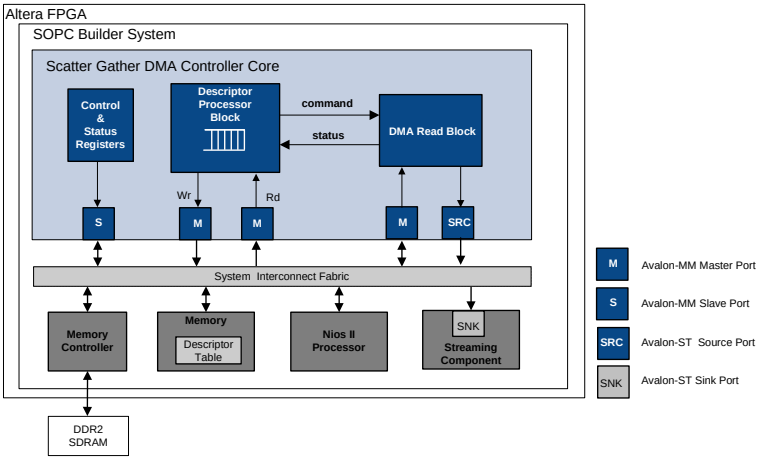


Memory-to-Stream Configuration

Memory-to-stream configurations include the descriptor processor and DMA read blocks. Figure 24–4 illustrates a memory-to-stream configuration.

In this example, the Nios II processor and descriptor table are in the FPGA. Data from an external DDR2 SDRAM is read by the SG-DMA controller and written to an on-chip streaming peripheral.

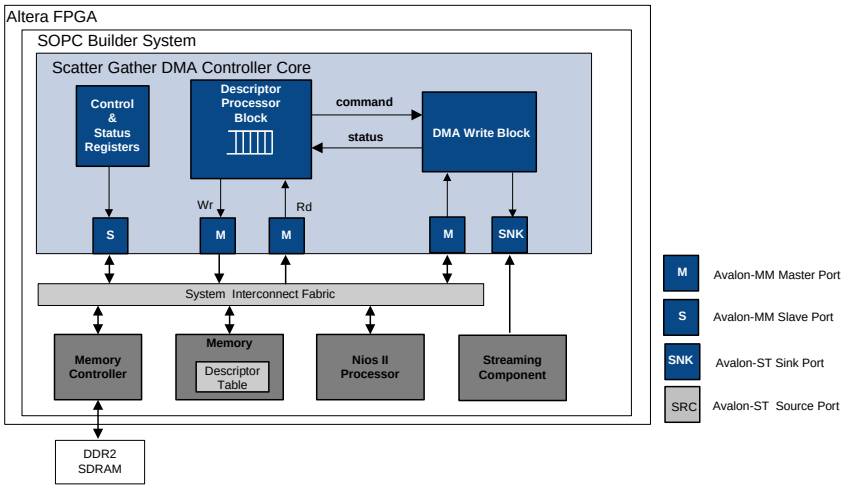
Figure 24–4. Example of Memory-to-Stream Configuration



Stream-to-Memory Configuration

Stream-to-memory configurations include the descriptor processor and DMA write blocks. This configuration is similar to the memory-to-stream configuration as Figure 24–5 illustrates.

Figure 24–5. Example of Memory-to-Stream Configuration



DMA Descriptors

DMA descriptors specify data transfers to be performed. The SG-DMA core uses a dedicated interface to read and write the descriptors. These descriptors, which are stored as a linked list, can be stored on an on-chip or off-chip memory and can be arbitrarily long.

Storing the descriptor list in an external memory frees up resources in the FPGA; however, an external descriptor list increases the overhead involved when the descriptor processor reads and updates the list. The SG-DMA core has an internal FIFO to store descriptors read from memory, which allows the core to perform descriptor read, execute, and write back operations in parallel, hiding the descriptor access and processing overhead.

☞ The descriptors must be initialized and aligned on a 32-bit boundary. The last descriptor in the list must have its `OWNED_BY_HW` bit set to 0 because the core relies on a cleared `OWNED_BY_HW` bit to stop processing.

See "DMA Descriptors" on page 24–13 for the structure of the DMA descriptor.

Descriptor Processing

The following steps describe how the DMA descriptors are processed:

1. Software builds the descriptor linked list. See "Building and Updating Descriptor List" on page 24–8 for more information on how to build and update the descriptor linked list.
2. Software writes the address of the first descriptor to the `next_descriptor_pointer` register and initiates the transfer by setting the `RUN` bit in the `control` register to 1. See "Software Programming Model" on page 24–10 for more information on the registers.

   On the next clock cycle following the assertion of the `RUN` bit, the core sets the `BUSY` bit in the `status` register to 1 to indicate that descriptor processing is executing.
3. The descriptor processor block reads the address of the first descriptor from the `next_descriptor_pointer` register and pushes the retrieved descriptor into the command FIFO, which feeds commands to both the DMA read and write blocks. As soon as the first descriptor is read, the block reads the next descriptor and pushes it into the command FIFO. One descriptor is always read in advance thus maximizing throughput.
4. The core performs the data transfer.
   - In memory-to-memory configurations, the DMA read block receives the source address from its command FIFO and starts reading data to fill the FIFO on its stream port until the specified number of bytes are transferred. The DMA read block pauses when the FIFO is full until the FIFO has enough space to accept more data.

     The DMA write block gets the destination address from its command FIFO and starts writing until the specified number of bytes are transferred. If the data FIFO ever empties, the write block pauses until the FIFO has more data to write.
   - In memory-to-stream configurations, the DMA read block reads from the source address and transfers the data to the core's streaming port until the specified number of bytes are transferred or the end of packet is reached. The block uses the end-of-packet indicator for transfers with an unknown transfer size. For data transfers without using the end-of-packet indicator, the transfer size must be a multiple of the data width. Otherwise, the block requires extra logic and may impact the system performance.
   - In stream-to-memory configurations, the DMA write block reads from the core's streaming port and writes to the destination address. The block continues reading until the specified number of bytes are transferred.
5. The descriptor processor block receives a status from the DMA read or write block and updates the `DESC_CONTROL`, `DESC_STATUS`, and `ACTUAL_BYTES_TRANSFERRED` fields in the descriptor. The `OWNED_BY_HW` bit in the `DESC_CONTROL` field is cleared unless the `PARK` bit is set to 1.

Once the core starts processing the descriptors, software must not update descriptors with `OWNED_BY_HW` bit set to 1. It is only safe for software to update a descriptor when its `OWNED_BY_HW` bit is cleared.

The SG-DMA core continues processing the descriptors until an error condition occurs and the `STOP_DMA_ER` bit is set to 1, or a descriptor with a cleared `OWNED_BY_HW` bit is encountered.

Building and Updating Descriptor List

Altera recommends the following method of building and updating the descriptor list:

1. Build the descriptor list and terminate the list with a non-hardware owned descriptor (`OWNED_BY_HW` = 0). The list can be arbitrarily long.
2. Set the interrupt `IE_CHAIN_COMPLETED`.
3. Write the address of the first descriptor in the first list to the `next_descriptor_pointer` register and set the `RUN` bit to 1 to initiate transfers.
4. While the core is processing the first list, build a second list of descriptors.
5. When the SD-DMA controller core finishes processing the first list, an interrupt is generated. Update the `next_descriptor_pointer` register with the address of the first descriptor in the second list. Clear the `RUN` bit and the `status` register. Set the `RUN` bit back to 1 to resume transfers.
6. If there are new descriptors to add, always add them to the list which the core is not processing. For example, if the core is processing the first list, add new descriptors to the second list and so forth.

This method ensures that the descriptors are not updated when the core is processing them. Because the method requires a response to the interrupt, a high-latency interrupt may cause a problem in systems where stalling data movement is not possible.

Error Conditions

The SG-DMA core has a configurable error width. Error signals are connected directly to the Avalon-ST source or sink to which the SG-DMA core is connected.

The list below describes how the error signals in the SG-DMA core are implemented in the folowing configurations:

- Memory-to-memory configuration

  No error signals are generated. The error field in the register and descriptor is hardcoded to 0.

- Memory-to-stream configuration

  If you specified the usage of error bits in the core, the error bits are generated in the Avalon-ST source interface. These error bits are hardcoded to 0 and generated in compliance with the Avalon-ST slave interfaces.

- Stream-to-memory configuration

  If you specified the usage of error bits in the core, error bits are generated in the Avalon-ST sink interface. These error bits are passed from the Avalon-ST sink interface and stored in the registers and descriptor.

Table 24–3 lists the error signals when the core is operating in the memory-to-stream configuration and connected to the transmit FIFO interface of the Altera Triple-Speed Ethernet MegaCore® function.

Table 24–3. Avalon-ST Transmit Error Types

Signal Type	Description
TSE_transmit_error[0]	Transmit Frame Error. Asserted to indicate that the transmitted frame should be viewed as invalid by the Ethernet MAC. The frame is then transferred onto the GMII interface with an error code during the frame transfer.

Table 24–4 lists the error signals when the core is operating in the stream-to-memory configuration and connected to the transmit FIFO interface of the Triple-Speed Ethernet MegaCore function.

Table 24–4. Avalon-ST Receive Error Types

Signal Type	Description
TSE_receive_error[0]	Receive Frame Error. This signal indicates that an error has occurred. It is the logical OR of receive errors 1 through 5.
TSE_receive_error[1]	Invalid Length Error. Asserted when the received frame has an invalid length as defined by the IEEE 802.3 standard.
TSE_receive_error[2]	CRC Error. Asserted when the frame has been received with a CRC-32 error.
TSE_receive_error[3]	Receive Frame Truncated. Asserted when the received frame has been truncated due to receive FIFO overflow.
TSE_receive_error[4]	Received Frame corrupted due to PHY error. (The PHY has asserted an error on the receive GMII interface.)
TSE_receive_error[5]	Collision Error. Asserted when the frame was received with a collision.

Each streaming core has a different set of error codes. Refer to the respective user guides for the codes.

Device Support

The SG-DMA Controller core supports all Altera device families.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the SG-DMA Controller core in SOPC Builder to add the core to a system.

☞ The SG-DMA controller core should be given a higher priority (lower IRQ value) than most of the components in a system to ensure high throughput.

Table 24–5 lists and describes the parameters you can configure.

Table 24–5. Configurable Parameters

Parameter	Legal Values	Description
Transfer mode	Memory To Memory Memory To Stream Stream To Memory	Configuration to use. For more information about these configurations, see "Memory-to-Memory Configuration" on page 24–5
Enable bursting on descriptor read master	On/Off	If this option is on, the descriptor processor block uses Avalon-MM bursting when fetching descriptors and writing them back in memory. With 32-bit read and write ports, the descriptor processor block can fetch the 256-bit descriptor by performing 8-word burst as opposed to eight individual single-word transactions.
Allow unaligned transfers	On/Off	If this option is on, the core allows accesses to non-word-aligned addresses. This option doesn't apply for burst transfers. Unaligned transfers require extra logic that may negatively impact system performance.
Enable burst transfers	On/Off	Turning on this option enables burst reads and writes.
Read burstcount signal width	1–16	The width of the read `burstcount` signal. This value determines the maximum burst read size.
Write burstcount signal width	1–16	The width of the write `burstcount` signal. This value determines the maximum burst write size.
Data width	8, 16, 32, 64	The data width in bits for the Avalon-MM read and write ports.
Source error width	0–7	The width of the `error` signal for the Avalon-ST source port.
Sink error width	0 – 7	The width of the `error` signal for the Avalon-ST sink port.
Data transfer FIFO depth	2, 4, 8, 16, 32, 64	The depth of the internal data FIFO in memory-to-memory configurations with burst transfers disabled.

Simulation Considerations

Signals for hardware simulation are automatically generated as part of the Nios II simulation process available in the Nios II IDE.

Software Programming Model

The following sections describe the software programming model for the SG-DMA controller core.

HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the SG-DMA controller core via the familiar HAL API and the ANSI C standard library.

Software Files

The SG-DMA controller core provides the following software files. These files provide low-level access to the hardware and drivers that integrate into the Nios II HAL system library. Application developers should not modify these files.

- **altera_avalon_sgdma_regs.h**—defines the core's register map, providing symbolic constants to access the low-level hardware

- **altera_avalon_sgdma.h**—provides definitions for the Altera Avalon SG-DMA buffer control and status flags.
- **altera_avalon_sgdma.c**—provides function definitions for the code that implements the SG-DMA controller core.
- **altera_avalon_sgdma_descriptor.h**—defines the core's descriptor, providing symbolic constants to access the low-level hardware.

Register Maps

The SG-DMA controller core has three registers accessible from its Avalon-MM interface; `status`, `control` and `next_descriptor_pointer`. Software can configure the core and determines its current status by accessing the registers.

The `control/status` register has a 32-bit interface without byte-enable logic, and therefore cannot be properly accessed by a master with narrower data width than itself. To ensure correct operation of the core, always access the register with a master that is at least 32 bits wide.

Table 24–6 lists and describes the registers.

Table 24–6. Register Map

32-bit Word Offset	Register Name	Reset Value	Description
base + 0	`status`	0	This register indicates the core's current status such as what caused the last interrupt and if the core is still processing descriptors. See Table 24–4 on page 24–9 for the `status` register map.
base + 4	`control`	0	This register specifies the core's behavior such as what triggers an interrupt and when the core is started and stopped. The host processor can configure the core by setting the register bits accordingly. See Table 24–4 on page 24–9 for the `control` register map.
base + 8	`next_descriptor_pointer`	0	This register contains the address of the next descriptor to process. Set this register to the address of the first descriptor as part of the system initialization sequence. Altera recommends that user applications clear the `RUN` bit in the `control` register and wait until the `BUSY` bit of the `status` register is set to 0 before reading this register.

Table 24–7 provides a bit map for the `control` register.

Table 24–7. Control Register Bit Map (Part 1 of 2)

Bit	Bit Name	Access	Description
0	`IE_ERROR`	R/W	When this bit is set to 1, the core generates an interrupt if an Avalon-ST error occurs during descriptor processing. *(1)*
1	`IE_EOP_ENCOUNTERED`	R/W	When this bit is set to 1, the core generates an interrupt if an EOP is encountered during descriptor processing. *(1)*
2	`IE_DESCRIPTOR_COMPLETED`	R/W	When this bit is set to 1, the core generates an interrupt after each descriptor is processed. *(1)*

Table 24–7. Control Register Bit Map (Part 2 of 2)

Bit	Bit Name	Access	Description
3	IE_CHAIN_COMPLETED	R/W	When this bit is set to 1, the core generates an interrupt after the last descriptor in the list is processed, that is when the core encounters a descriptor with a cleared OWNED_BY_HW bit. *(1)*
4	IE_GLOBAL	R/W	Global signal to enable all interrupts.
5	RUN	R/W	Set this bit to 1 to start the descriptor processor block which subsequently initiates DMA transactions. Prior to setting this bit to 1, ensure that the register next_descriptor_pointer is updated with the address of the first descriptor to process. The core continues to process descriptors in its queue as long as this bit is 1. Clear this bit to stop the core from processing the next descriptor in its queue. If this bit is cleared in the middle of processing a descriptor, the core completes the processing before stopping. The host processor can then modify the remaining descriptors and restart the core.
6	STOP_DMA_ER	R/W	Set this bit to 1 to stop the core when an Avalon-ST error is encountered during a DMA transaction. This bit applies only to stream-to-memory configurations.
7	IE_MAX_DESC_PROCESSED	R/W	Set this bit to 1 to generate an interrupt after the number of descriptors specified by MAX_DESC_PROCESSED are processed.
8 .. 15	MAX_DESC_PROCESSED	R/W	Specifies the number of descriptors to process before the core generates an interrupt.
16	SW_RESET	R/W	Software can reset the core by writing to this bit twice. Upon the second write, the core is reset. The logic which sequences the software reset process then resets itself automatically. Executing a software reset when a DMA transfer is active may result in permanent bus lockup until the next system reset. Hence, Altera recommends that you use the software reset as your last resort.
17	PARK	R/W	Seting this bit to 0 causes the SG-DMA controller core to clear the OWNED_BY_HW bit in the descriptor after each descriptor is processed. If the PARK bit is set to 1, the core does not clear the OWNED_BY_HW bit, thus allowing the same descriptor to be processed repeatedly without software intervention. You also need to set the last descriptor in the list to point to the first one.
18..30	Reserved		
31	CLEAR_INTERRUPT	R/W	Set this bit to 1 to clear pending interrupts.

Note to Table 24–7:

(1) All interrupts are generated only after the descriptor is updated.

Table 24–8 provides a bit map for the status register. Altera recommends that you read the status register only after the RUN bit in the control register is cleared.

Table 24–8. Status Register Bit Map

Bit	Bit Name	Access	Description
0	ERROR	R/C *(1) (2)*	A value of 1 indicates that an Avalon-ST error was encountered during a transfer.
1	EOP_ENCOUNTERED	R/C	A value of 1 indicates that the transfer was terminated by an end-of-packet (EOP) signal generated on the Avalon-ST source interface. This condition is only possible in stream-to-memory configurations.
2	DESCRIPTOR_COMPLETED	R/C *(1) (2)*	A value of 1 indicates that a descriptor was processed to completion.
3	CHAIN_COMPLETED	R/C *(1) (2)*	A value of 1 indicates that the core has completed processing the descriptor chain.
4	BUSY	R *(1) (3)*	A value of 1 indicates that descriptors are being processed. This bit is set to 1 on the next clock cycle after the RUN bit is asserted and does not get cleared until one of the following event occurs: ■ Descriptor processing completes and the RUN bit is cleared. ■ An error condition occurs, the STOP_DMA_ER bit is set to 1 and the processing of the current descriptor completes.
5 .. 31	Reserved		

Notes to Table 24–8:

(1) This bit must be cleared after a read is performed. Write one to clear this bit.

(2) This bit is updated by hardware after each DMA transfer completes. It remains set until software writes one to clear.

(3) This bit is continuously updated by the hardware.

DMA Descriptors

Table 24–9 shows the structure a DMA descriptor entry. See "Data Structure" on page 24–15 for the structure definition.

Table 24–9. DMA Descriptor Structure

Byte Offset	Field Names			
	31 24	23 16	15 8	7 0
base	source			
base + 4	Reserved			
base + 8	destination			
base + 12	Reserved			
base + 16	next_desc_ptr			
base + 20	Reserved			
base + 24	Reserved		bytes_to_transfer	
base + 28	desc_control	desc_status	actual_bytes_transferred	

Table 24–10 describes each field in a descriptor entry.

Table 24–10. DMA Descriptor Field Description

Field Name	Access	Description
`source`	R/W	Specifies the address of data to be read. This address is set to 0 if the input interface is an Avalon-ST interface.
`destination`	R/W	Specifies the address to which data should be written. This address is set to 0 if the output interface is an Avalon-ST interface.
`next_desc_ptr`	R/W	Specifies the address of the next descriptor in the linked list.
`bytes_to_transfer`	R/W	Specifies the number of bytes to transfer. If this field is 0, the SG-DMA controller core continues transferring data until it encounters an EOP.
`actual_bytes_transferred`	R	Specifies the number of bytes that are successfully transferred by the core. This field is updated after the core processes a descriptor.
`desc_status`	R/W	This field is updated after the core processes a descriptor. See Table 24–12 on page 24–15 for the bit map of this field.
`desc_control`	R/W	Specifies the behavior of the core. This field is updated after the core processes a descriptor. See Table 24–11 on page 24–14 for descriptions of each bit.

Table 24–11 provides a bit map for the `desc_control` field.

Table 24–11. DESC_CONTROL Bit Map

Bit (s)	Field Name	Access	Description
0	`GENERATE_EOP`	W	When this bit is set to 1,the DMA read block asserts the EOP signal on the final word.
1	`READ_FIXED_ADDRESS`	R/W	This bit applies only to Avalon-MM read master ports. When this bit is set to 1, the DMA read block does not increment the memory address. When this bit is set to 0, the read address increments after each read.
2	`WRITE_FIXED_ADDRESS`	R/W	This bit applies only to Avalon-MM write master ports. When this bit is set to 1, the DMA write block does not increment the memory address. When this bit is set to 0, the write address increments after each write. In memory-to-stream configurations, the DMA read block generates a start-of-packet (SOP) on the first word when this bit is set to 1.
[6:3]	Reserved	—	—
7	`OWNED_BY_HW`	R/W	This bit determines whether hardware or software has write access to the current register. When this bit is set to 1, the core can update the descriptor and software should not access the descriptor due to the possibility of race conditions. Otherwise, it is safe for software to update the descriptor.

After completing a DMA transaction, the descriptor processor block updates the `desc_status` field to indicate how the transaction proceeded. Table 24–1 provides the bit map of this field.

Table 24–12. DESC_STATUS Bit Map

Bit	Bit Name	Access	Description
[7:0]	ERROR_0 .. ERROR_7	R	Each bit represents an error that occurred on the Avalon-ST interface. The context of each error is defined by the component connected to the Avalon-ST interface.

Timeouts

The SG-DMA controller does not implement internal counters to detect stalls. Software can instantiate a timer component if this functionality is required.

Programming with SG-DMA Controller

This section describes the device and descriptor data structures, and the application programming interface (API) for the SG-DMA controller core.

Data Structure

Figure 24–6 shows the data structure for the device.

Figure 24–6. Device Data Structure

```
typedef struct alt_sgdma_dev
{
  alt_llist                 llist;               // Device linked-list entry
  const char                *name;               // Name of SGDMA in SOPC System
  void                      *base;               // Base address of SGDMA
  alt_u32                   *descriptor_base;    // reserved
  alt_u32                   next_index;          // reserved
  alt_u32                   num_descriptors;     // reserved
  alt_sgdma_descriptor      *current_descriptor; // reserved
  alt_sgdma_descriptor      *next_descriptor;    // reserved
  alt_avalon_sgdma_callback callback;            // Callback routine pointer
  void                      *callback_context;   // Callback context pointer
  alt_u32                   chain_control;       // Value OR'd into control reg
} alt_sgdma_dev;
```

Figure 24–7 shows the data structure for the descriptors.

Figure 24–7. Descriptor Data Structure

```
typedef struct {
    alt_u32   *read_addr;
    alt_u32   read_addr_pad;

    alt_u32   *write_addr;
    alt_u32   write_addr_pad;

    alt_u32   *next;
    alt_u32   next_pad;

    alt_u16   bytes_to_transfer;
    alt_u8    read_burst; /* Reserved field. Set to 0. */
    alt_u8    write_burst;/* Reserved field. Set to 0. */

    alt_u16   actual_bytes_transferred;
    alt_u8    status;
    alt_u8    control;

} alt_avalon_sgdma_packed alt_sgdma_descriptor;
```

SG-DMA API

Table 24–13 lists all functions provided and briefly describes each.

Table 24–13. Function List

Name	Description
`alt_avalon_sgdma_do_async_transfer()`	Starts a non-blocking transfer of a descriptor chain.
`alt_avalon_sgdma_do_sync_transfer()`	Starts a blocking transfer of a descriptor chain. This function blocks both before transfer if the controller is busy and until the requested transfer has completed.
`alt_avalon_sgdma_construct_mem_to_mem_desc()`	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-MM transfer.
`alt_avalon_sgdma_construct_stream_to_mem_desc()`	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-ST to Avalon-MM transfer. The function automatically terminates the descriptor chain with a NULL descriptor.
`alt_avalon_sgdma_construct_mem_to_stream_desc()`	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-ST transfer.
`alt_avalon_sgdma_check_descriptor_status()`	Reads the status of a given descriptor.
`alt_avalon_sgdma_register_callback()`	Associates a user-specific callback routine with the SG-DMA interrupt handler.
`alt_avalon_sgdma_start()`	Starts the DMA engine. This is not required when `alt_avalon_sgdma_do_async_transfer()` and `alt_avalon_sgdma_do_sync_transfer()` are used.
`alt_avalon_sgdma_stop()`	Stops the DMA engine. This is not required when `alt_avalon_sgdma_do_async_transfer()` and `alt_avalon_sgdma_do_sync_transfer()` are used.
`alt_avalon_sgdma_open()`	Returns a pointer to the SG-DMA controller with the given name.

alt_avalon_sgdma_do_async_transfer()

Prototype: int alt_avalon_do_async_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc)

Thread-safe: No.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*dev`—a pointer to an SG-DMA device structure.

`*desc`—a pointer to a single, constructed descriptor. The descriptor must have its "next" descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain.

Returns: Returns 0 success. Other return codes are defined in **errno.h**.

Description: Set up and begin a non-blocking transfer of one or more descriptors or a descriptor chain. If the SG-DMA controller is busy at the time of this call, the routine immediately returns `EBUSY`; the application can then decide how to proceed without being blocked. If a callback routine has been previously registered with this particular SG-DMA controller, the transfer is set up to issue an interrupt on error, EOP, or chain completion. Otherwise, no interrupt is registered and the application developer must check for and handle errors and completion. The run bit is cleared before the begining of the transfer and is set to 1 to restart a new descriptor chain.

alt_avalon_sgdma_do_sync_transfer()

Prototype: alt_u8 alt_avalon_sgdma_do_sync_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc)

Thread-safe: No.

Available from ISR: Not recommended.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*dev`—a pointer to an SG-DMA device structure.

`*desc`—a pointer to a single, constructed descriptor. The descriptor must have its "next" descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain.

Returns: Returns the contents of the `status` register.

Description: Sends a fully formed descriptor or list of descriptors to the SG-DMA controller for transfer. This function blocks both before transfer, if the SG-DMA controller is busy, and until the requested transfer has completed. If an error is detected during the transfer, it is abandoned and the controller's `status` register contents are returned to the caller. Additional error information is available in the status bits of each descriptor that the SG-DMA processed. The user application searches through the descriptor or list of descriptors to gather specific error information. The run bit is cleared before the begining of the transfer and is set to 1 to restart a new descriptor chain.

alt_avalon_sgdma_construct_mem_to_mem_desc()

Prototype:	void alt_avalon_sgdma_construct_mem_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr, alt_u32 *write_addr, alt_u16 length, int read_fixed, int write_fixed)
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*desc`—a pointer to the descriptor being constructed. `*next`—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated. `*read_addr`—the first read address for the SG-DMA transfer. `*write_addr`—the first write address for the SG-DMA transfer. `length`—the number of bytes for the transfer. `read_fixed`—if non-zero, the SG-DMA reads from a fixed address. `write_fixed`—if non-zero, the SG-DMA writes to a fixed address.
Returns:	void
Description:	This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma_descriptor *desc` for an Avalon-MM to Avalon-MM transfer. The function sets the `OWNED_BY_HW` bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the `RUN` bit is 1. The next field of the descriptor being constructed is set to the address in `*next`. The `OWNED_BY_HW` bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it does not process the descriptor at `*next` until its `OWNED_BY_HW` bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter. You must properly allocate memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device mastered by the SG-DMA controller's chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller.

alt_avalon_sgdma_construct_stream_to_mem_desc()

Prototype: void alt_avalon_sgdma_construct_stream_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *write_addr, alt_u16 length_or_eop, int write_fixed)

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*desc`—a pointer to the descriptor being constructed.

`*next`—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.

`*write_addr`—the first write address for the SG-DMA transfer.

`length_or_eop`—the number of bytes for the transfer. If set to zero (0x0), the transfer continues until an EOP signal is received from the Avalon-ST interface.

`write_fixed`—if non-zero, the SG-DMA will write to a fixed address.

Returns: void

Description: This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma_descriptor *desc` for an Avalon-ST to Avalon-MM transfer. The source (read) data for the transfer comes from the Avalon-ST interface connected to the SG-DMA controller's streaming read port.

The function sets the `OWNED_BY_HW` bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the `RUN` bit is 1.

The next field of the descriptor being constructed is set to the address in `*next`. The `OWNED_BY_HW` bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it does not process the descriptor at `*next` until its `OWNED_BY_HW` bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter.

You must properly allocate memory for the creation of both the descriptor under construction as well as the next descriptor in the chain.

Descriptors must be in a memory device mastered by the SG-DMA controller's chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller.

alt_avalon_sgdma_construct_mem_to_stream_desc()

Prototype: void alt_avalon_sgdma_construct_mem_to_stream_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr, alt_u16 length, int read_fixed, int generate_sop, int generate_eop, alt_u8 atlantic_channel)

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*desc`—a pointer to the descriptor being constructed.

`*next`—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.

`*read_addr`—the first read address for the SG-DMA transfer.

`length`—the number of bytes for the transfer.

`read_fixed`—if non-zero, the SG-DMA reads from a fixed address.

`generate_sop`—if non-zero, the SG-DMA generates a SOP on the Avalon-ST interface when commencing the transfer.

`generate_eop`—if non-zero, the SG-DMA generates an EOP on the Avalon-ST interface when completing the transfer.

`atlantic_channel`—an 8-bit Avalon-ST channel number. Channels are currently not supported. Set this parameter to 0.

Returns: void

Description: This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma-descriptor *desc` for an Avalon-MM to Avalon-ST transfer. The destination (write) data for the transfer goes to the Avalon-ST interface connected to the SG-DMA controller's streaming write port. The function sets the `OWNED_BY_HW` bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the `RUN` bit is 1.

The next field of the descriptor being constructed is set to the address in `*next`. The `OWNED_BY_HW` bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it does not process the descriptor at `*next` until its `OWNED_BY_HW` bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter.

You are responsible for properly allocating memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device mastered by the SG-DMA controller's chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller.

alt_avalon_sgdma_check_descriptor_status()

Prototype: int alt_avalon_sgdma_check_descriptor_status(alt_sgdma_descriptor *desc)

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*desc`—a pointer to the constructed descriptor to examine.

Returns: Returns 0 if the descriptor is error-free, not owned by hardware, or a previously requested transfer completed normally. Other return codes are defined in **errno.h**.

Description: Checks a descriptor previously owned by hardware for any errors reported in a previous transfer. The routine reports: errors reported by the SG-DMA controller, the buffer in use.

alt_avalon_sgdma_register_callback()

Prototype: void alt_avalon_sgdma_register_callback(alt_sgdma_dev *dev, alt_avalon_sgdma_callback callback, alt_u16 chain_control, void *context)

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*dev`—a pointer to the SG-DMA device structure.

`callback`—a pointer to the callback routine to execute at interrupt level.

`chain_control`—the SG-DMA control register contents.

`*context`—a pointer used to pass context-specific information to the ISR. `context` can point to any ISR-specific information.

Returns: void

Description: Associates a user-specific routine with the SG-DMA interrupt handler. If a callback is registered, all non-blocking transfers enables interrupts that causes the callback to be executed. The callback runs as part of the interrupt service routine, and care must be taken to follow the guidelines for acceptable interrupt service routine behavior as described in the *Nios II Software Developer's Handbook*.

To disable callbacks after registering one, call this routine with 0x0 as the callback argument.

alt_avalon_sgdma_start()

Prototype: void alt_avalon_sgdma_start(alt_sgdma_dev *dev)

Thread-safe: No.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*dev`—a pointer to the SG-DMA device structure.

Returns: void

Description: Starts the DMA engine and processes the descriptor pointed to in the controller's next descriptor pointer and all subsequent descriptors in the chain. It is not necessary to call this function when `do_sync` or `do_async` is used.

alt_avalon_sgdma_stop()

Prototype:	void alt_avalon_sgdma_stop(alt_sgdma_dev *dev)
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*dev`—a pointer to the SG-DMA device structure.
Returns:	void
Description:	Stops the DMA engine following completion of the current buffer descriptor. It is not necessary to call this function when `do_sync` or `do_async` is used.

alt_avalon_sgdma_open()

Prototype:	alt_sgdma_dev* alt_avalon_sgdma_open(const char* name)
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`name`—the name of the SG-DMA device to open.
Returns:	A pointer to the SG-DMA device structure associated with the supplied name, or NULL if no corresponding SG-DMA device structure was found.
Description:	Retrieves a pointer to a hardware SG-DMA device structure.

Referenced Documents

This chapter references the *Nios II Software Developer's Handbook*.

Document Revision History

Table 24–14 shows the revision history for this chapter.

Table 24–14. Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	■ Revised descriptions of register fields and bits. ■ Added description to the memory-to-stream configurations. ■ Added descriptions to alt_avalon_sgdma_do_sync_transfer() and alt_avalon_sgdma_do_async_transfer() API. ■ Added a list on error signals implementation.	—
March 2009 v9.0.0	Added description of **Enable bursting on descriptor read master**.	—
November 2008 v8.1.0	■ Changed to 8-1/2 x 11 page size. ■ Added section DMA Descriptors in Functional Specifications ■ Revised descriptions of register fields and bits. ■ Reorganized sections Software Programming Model and Programming with SG-DMA Controller Core.	—
May 2008 v8.0.0	■ Added sections on burst transfers.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

25. DMA Controller Core

Core Overview

The direct memory access (DMA) controller core with Avalon® interface performs bulk data transfers, reading data from a source address range and writing the data to a different address range. An Avalon Memor-Mapped (Avalon-MM) master peripheral, such as a CPU, can offload memory transfer tasks to the DMA controller. While the DMA controller performs memory transfers, the master is free to perform other tasks in parallel.

The DMA controller transfers data as efficiently as possible, reading and writing data at the maximum pace allowed by the source or destination. The DMA controller is capable of performing Avalon transfers with flow control, enabling it to automatically transfer data to or from a slow peripheral with flow control (for example, UART), at the maximum pace allowed by the peripheral.

The DMA controller is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios® II processor, device drivers are provided in the HAL system library. See "Software Programming Model" on page 25–5 for details of HAL support.

This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 25–4
- "Device Support" on page 25–5
- "Software Programming Model" on page 25–5

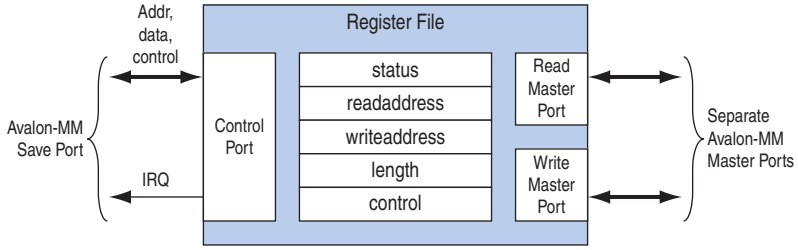
Functional Description

You can use the DMA controller to perform data transfers from a source address-space to a destination address-space. The controller has no concept of endianness and does not interpret the payload data. The concept of endianness only applies to a master that interprets payload data.

The source and destination may be either an Avalon-MM slave peripheral (for example, a constant address) or an address range in memory. The DMA controller can be used in conjunction with peripherals with flow control, which allows data transactions of fixed or variable length. The DMA controller can signal an interrupt request (IRQ) when a DMA transaction completes. A transaction is a sequence of one or more Avalon transfers initiated by the DMA controller core.

The DMA controller has two Avalon-MM master ports—a master read port and a master write port—and one Avalon-MM slave port for controlling the DMA as shown in Figure 25–1.

Figure 25–1. DMA Controller Block Diagram



A typical DMA transaction proceeds as follows:

1. A CPU prepares the DMA controller for a transaction by writing to the control port.
2. The CPU enables the DMA controller. The DMA controller then begins transferring data without additional intervention from the CPU. The DMA's master read port reads data from the read address, which may be a memory or a peripheral. The master write port writes the data to the destination address, which can also be a memory or peripheral. A shallow FIFO buffers data between the read and write ports.
3. The DMA transaction ends when a specified number of bytes are transferred (a fixed-length transaction) or an end-of-packet signal is asserted by either the sender or receiver (a variable-length transaction). At the end of the transaction, the DMA controller generates an interrupt request (IRQ) if it was configured by the CPU to do so.
4. During or after the transaction, the CPU can determine if a transaction is in progress, or if the transaction ended (and how) by examining the DMA controller's `status` register.

Setting Up DMA Transactions

An Avalon-MM master peripheral sets up and initiates DMA transactions by writing to registers via the control port. The Avalon-MM master programs the DMA engine using byte addresses which are byte aligned. The master peripheral configures the following options:

- Read (source) address location
- Write (destination) address location
- Size of the individual transfers: Byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit) or quadword (128-bit)
- Enable interrupt upon end of transaction
- Enable source or destination to end the DMA transaction with end-of-packet signal
- Specify whether source and destination are memory or peripheral

The master peripheral then sets a bit in the `control` register to initiate the DMA transaction.

The Master Read and Write Ports

The DMA controller reads data from the source address through the master read port, and then writes to the destination address through the master write port. You program the DMA controller using byte addresses. Read and write start addresses should be aligned to the transfer size. For example, to transfer data words, if the start address is 0, the address will increment to 4, 8, and 12. For heterogeneous systems where a number of different slave devices are of different widths, the data width for read and write masters matches the width of the widest data-width slave addressed by either the read or the write master. For bursting transfers, the burst length is set to the DMA transaction length with the appropriate unit conversion. For example, if a 32-bit data width DMA is programmed for a word transfer of 64 bytes, the length registered is programmed with 64 and the burst count port will be 16. If a 64-bit data width DMA is programmed for a doubleword transfer of 8 bytes, the length register is programmed with 8 and the burst count port will be 1.

There is a shallow FIFO buffer between the master read and write ports. The default depth is 2, which makes the write action depend on the data-available status of the FIFO, rather than on the status of the master read port.

Both the read and write master ports can perform Avalon transfers with flow control, which allows the slave peripheral to control the flow of data and terminate the DMA transaction.

For details about flow control in Avalon-MM data transfers and Avalon-MM peripherals, refer to *Avalon Interface Specifications*.

Addressing and Address Incrementing

When accessing memory, the read (or write) address increments by 1, 2, 4, 8, or 16 after each access, depending on the width of the data. On the other hand, a typical peripheral device (such as UART) has fixed register locations. In this case, the read/write address is held constant throughout the DMA transaction.

The rules for address incrementing are, in order of priority:

- If the `control` register's `RCON` (or `WCON`) bit is set, the read (or write) increment value is 0.
- Otherwise, the read and write increment values are set according to the transfer size specified in the control register, as shown in Table 25–1.

Table 25–1. Address Increment Values

Transfer Width	Increment
byte	1
halfword	2
word	4
doubleword	8
quadword	16

In systems with heterogeneous data widths, care must be taken to present the correct address or offset when configuring the DMA to access native-aligned slaves. For example, in a system using a 32-bit Nios II processor and a 16-bit DMA, the base address for the UART `txdata` register must be divided by the dma_data_width/cpu_data_width—2 in this example.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the DMA controller in SOPC Builder to specify the core's configuration. Instantiating the DMA controller in SOPC Builder creates one slave port and two master ports. You must specify which slave peripherals can be accessed by the read and write master ports. Likewise, you must specify which other master peripheral(s) can access the DMA control port and initiate DMA transactions. The DMA controller does not export any signals to the top level of the system module.

DMA Parameters (Basic)

This section describes the parameters you can configure on the **DMA Parameters** page.

Transfer Size

The parameter **Width of the DMA Length Register** specifies the minimum width of the DMA's transaction length register, which can be between 1 and 32. The `length` register determines the maximum number of transfers possible in a single DMA transaction.

By default, the length register is wide enough to span any of the slave peripherals mastered by the read or write ports. Overriding the length register may be necessary if the DMA master port (read or write) masters only data peripherals, such as a UART. In this case, the address span of each slave is small, but a larger number of transfers may be desired per DMA transaction.

Burst Transactions

When **Enable Burst Transfers** is turned on, the DMA controller performs burst transactions on its master read and write ports. The parameter **Maximum Burst Size** determines the maximum burst size allowed in a transaction.

In burst mode, the length of a transaction must not be longer than the configured maximum burst size. Otherwise, the transaction must be performed as multiple transactions.

FIFO Depth

The parameter **Data Transfer FIFO Depth** specifies the depth of the FIFO buffer used for data transfers. Altera recommends that you set the depth of the FIFO buffer to at least twice the maximum read latency of the slave interface connected to the read master port. A depth that is too low reduces transfer throughput.

FIFO Implementation

This option determines the implementation of the FIFO buffer between the master read and write ports. Select **Construct FIFO from Registers** to implement the FIFO using one register per storage bit. This option has a strong impact on logic utilization when the DMA controller's data width is large. See "Advanced Options" on page 25–5.

To implement the FIFO using embedded memory blocks available in the FPGA, select **Construct FIFO from Memory Blocks**.

Advanced Options

This section describes the parameters you can configure on the **Advanced Options** page.

Allowed Transactions

You can choose the transfer datawidth(s) supported by the DMA controller hardware. The following datawidth options can be enabled or disabled:

- Byte
- Halfword (two bytes)
- Word (four bytes)
- Doubleword (eight bytes)
- Quadword (sixteen bytes)

Disabling unnecessary transfer widths reduces the number of on-chip logic resources consumed by the DMA controller core. For example, if a system has both 16-bit and 32-bit memories, but the DMA controller transfers data to the 16-bit memory, 32-bit transfers could be disabled to conserve logic resources.

Device Support

The DMA Controller Core with Avalon Interface supports all Altera device families.

Software Programming Model

This section describes the programming model for the DMA controller, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the DMA controller core using the HAL API for DMA devices.

HAL System Library Support

The Altera-provided driver implements a HAL DMA device driver that integrates into the HAL system library for Nios II systems. HAL users should access the DMA controller via the familiar HAL API, rather than accessing the registers directly.

CAUTION

If your program uses the HAL device driver to access the DMA controller, accessing the device registers directly interferes with the correct behavior of the driver.

The HAL DMA driver provides both ends of the DMA process; the driver registers itself as both a receive channel (`alt_dma_rxchan`) and a transmit channel (`alt_dma_txchan`). The *Nios II Software Developer's Handbook* provides complete details of the HAL system library and the usage of DMA devices.

ioctl() Operations

`ioctl()` operation requests are defined for both the receive and transmit channels, which allows you to control the hardware-dependent aspects of the DMA controller. Two `ioctl()` functions are defined for the receiver driver and the transmitter driver: `alt_dma_rxchan_ioctl()` and `alt_dma_txchan_ioctl()`. Table 25–2 lists the available operations. These are valid for both the transmit and receive channels.

Table 25–2. Operations for alt_dma_rxchan_ioctl() and alt_dma_txchan_ioctl()

Request	Meaning
`ALT_DMA_SET_MODE_8`	Transfers data in units of 8 bits. The parameter `arg` is ignored.
`ALT_DMA_SET_MODE_16`	Transfers data in units of 16 bits. The parameter `arg` is ignored.
`ALT_DMA_SET_MODE_32`	Transfers data in units of 32 bits. The parameter `arg` is ignored.
`ALT_DMA_SET_MODE_64`	Transfers data in units of 64 bits. The parameter `arg` is ignored.
`ALT_DMA_SET_MODE_128`	Transfers data in units of 128 bits. The parameter `arg` is ignored.
`ALT_DMA_RX_ONLY_ON` *(1)*	Sets a DMA receiver into streaming mode. In this case, data is read continuously from a single location. The parameter `arg` specifies the address to read from.
`ALT_DMA_RX_ONLY_OFF` *(1)*	Turns off streaming mode for a receive channel. The parameter `arg` is ignored.
`ALT_DMA_TX_ONLY_ON` *(1)*	Sets a DMA transmitter into streaming mode. In this case, data is written continuously to a single location. The parameter `arg` specifies the address to write to.
`ALT_DMA_TX_ONLY_OFF` *(1)*	Turns off streaming mode for a transmit channel. The parameter `arg` is ignored.

Note to Table 25–2:

(1) These macro names changed in version 1.1 of the Nios II Embedded Design Suite (EDS). The old names (`ALT_DMA_TX_STREAM_ON`, `ALT_DMA_TX_STREAM_OFF`, `ALT_DMA_RX_STREAM_ON`, and `ALT_DMA_RX_STREAM_OFF`) are still valid, but new designs should use the new names.

Limitations

Currently the Altera-provided drivers do not support 64-bit and 128-bit DMA transactions.

This function is not thread safe. If you want to access the DMA controller from more than one thread then you should use a semaphore or mutex to ensure that only one thread is executing within this function at any time.

Software Files

The DMA controller is accompanied by the following software files. These files define the low-level interface to the hardware. Application developers should not modify these files.

- **altera_avalon_dma_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **altera_avalon_dma.h**, **altera_avalon_dma.c**—These files implement the DMA controller's device driver for the HAL system library.

Register Map

Programmers using the HAL API never access the DMA controller hardware directly via its registers. In general, the register map is only useful to programmers writing a device driver.

CAUTION

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 25–3 shows the register map for the DMA controller. Device drivers control and communicate with the hardware through five memory-mapped 32-bit registers.

Table 25–3. DMA Controller Register Map

Offset	Register Name	Read/Write	31 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	status *(1)*	RW	*(2)*									LEN	WEOP	REOP	BUSY	DONE
1	readaddress	RW	Read master start address													
2	writeaddress	RW	Write master start address													
3	length	RW	DMA transaction length (in bytes)													
4	—	—	Reserved *(3)*													
5	—	—	Reserved *(3)*													
6	control	RW	*(2)*	SOFTWARERESET	QUADWORD	DOUBLEWORD	WCON	RCON	LEEN	WEEN	REEN	I_EN	GO	WORD	HW	BYTE
7	—	—	Reserved *(3)*													

Notes to Table 25–3:

(1) Writing zero to the status register clears the LEN, WEOP, REOP, and DONE bits.

(2) These bits are reserved. Read values are undefined. Write zero.

(3) This register is reserved. Read values are undefined. The result of a write is undefined.

status Register

The status register consists of individual bits that indicate conditions inside the DMA controller. The status register can be read at any time. Reading the status register does not change its value.

The status register bits are shown in Table 25–4.

Table 25–4. status Register Bits (Part 1 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
0	DONE	R/C	A DMA transaction is complete. The DONE bit is set to 1 when an end of packet condition is detected or the specified transaction length is completed. Write zero to the status register to clear the DONE bit.
1	BUSY	R	The BUSY bit is 1 when a DMA transaction is in progress.

Table 25–4. status Register Bits (Part 2 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
2	REOP	R	The REOP bit is 1 when a transaction is completed due to an end-of-packet event on the read side.
3	WEOP	R	The WEOP bit is 1 when a transaction is completed due to an end of packet event on the write side.
4	LEN	R	The LEN bit is set to 1 when the length register decrements to zero.

readaddress Register

The `readaddress` register specifies the first location to be read in a DMA transaction. The `readaddress` register width is determined at system generation time. It is wide enough to address the full range of all slave ports mastered by the read port.

writeaddress Register

The `writeaddress` register specifies the first location to be written in a DMA transaction. The `writeaddress` register width is determined at system generation time. It is wide enough to address the full range of all slave ports mastered by the write port.

length Register

The `length` register specifies the number of bytes to be transferred from the read port to the write port. The `length` register is specified in bytes. For example, the value must be a multiple of 4 for word transfers, and a multiple of 2 for halfword transfers.

The `length` register is decremented as each data value is written by the write master port. When `length` reaches 0 the `LEN` bit is set. The `length` register does not decrement below 0.

The length register width is determined at system generation time. It is at least wide enough to span any of the slave ports mastered by the read or write master ports, and it can be made wider if necessary.

control Register

The control register is composed of individual bits that control the DMA's internal operation. The control register's value can be read at any time. The control register bits determine which, if any, conditions of the DMA transaction result in the end of a transaction and an interrupt request.

The control register bits are shown in Table 25–5.

Table 25–5. Control Register Bits (Part 1 of 2)

Bit Number	Bit Name	Read/ Write/ Clear	Description
0	BYTE	RW	Specifies byte transfers.
1	HW	RW	Specifies halfword (16-bit) transfers.
2	WORD	RW	Specifies word (32-bit) transfers.

Table 25–5. Control Register Bits (Part 2 of 2)

Bit Number	Bit Name	Read/ Write/ Clear	Description
3	GO	RW	Enables DMA transaction. When the GO bit is set to 0, the DMA is prevented from executing transfers. When the GO bit is set to 1 and the length register is non-zero, transfers occur.
4	I_EN	RW	Enables interrupt requests (IRQ). When the I_EN bit is 1, the DMA controller generates an IRQ when the status register's DONE bit is set to 1. IRQs are disabled when the I_EN bit is 0.
5	REEN	RW	Ends transaction on read-side end-of-packet. When the REEN bit is set to 1, a slave port with flow control on the read side may end the DMA transaction by asserting its end-of-packet signal.
6	WEEN	RW	Ends transaction on write-side end-of-packet. When the WEEN bit is set to 1, a slave port with flow control on the write side may end the DMA transaction by asserting its end-of-packet signal.
7	LEEN	RW	Ends transaction when the length register reaches zero. When the LEEN bit is 1, the DMA transaction ends when the length register reaches 0. When this bit is 0, length reaching 0 does not cause a transaction to end. In this case, the DMA transaction must be terminated by an end-of-packet signal from either the read or write master port.
8	RCON	RW	Reads from a constant address. When RCON is 0, the read address increments after every data transfer. This is the mechanism for the DMA controller to read a range of memory addresses. When RCON is 1, the read address does not increment. This is the mechanism for the DMA controller to read from a peripheral at a constant memory address. For details, see "Addressing and Address Incrementing" on page 25–3.
9	WCON	RW	Writes to a constant address. Similar to the RCON bit, when WCON is 0 the write address increments after every data transfer; when WCON is 1 the write address does not increment. For details, see "Addressing and Address Incrementing" on page 25–3.
10	DOUBLEWORD	RW	Specifies doubleword transfers.
11	QUADWORD	RW	Specifies quadword transfers.
12	SOFTWARERESET	RW	Software can reset the DMA engine by writing this bit to 1 twice. Upon the second write of 1 to the SOFTWARERESET bit, the DMA control is reset identically to a system reset. The logic which sequences the software reset process then resets itself automatically.

The data width of DMA transactions is specified by the BYTE, HW, WORD, DOUBLEWORD, and QUADWORD bits. Only one of these bits can be set at a time. If more than one of the bits is set, the DMA controller behavior is undefined. The width of the transfer is determined by the narrower of the two slaves read and written. For example, a DMA transaction that reads from a 16-bit flash memory and writes to a 32-bit on-chip memory requires a halfword transfer. In this case, HW must be set to 1, and BYTE, WORD, DOUBLEWORD, and QUADWORD must be set to 0.

To successfully perform transactions of a specific width, that width must be enabled in hardware using the **Allowed Transaction** hardware option. For example, the DMA controller behavior is undefined if quadword transfers are disabled in hardware, but the QUADWORD bit is set during a DMA transaction.

CAUTION Executing a DMA software reset when a DMA transfer is active may result in permanent bus lockup (until the next system reset). The `SOFTWARERESET` bit should therefore not be written except as a last resort.

Interrupt Behavior

The DMA controller has a single IRQ output that is asserted when the `status` register's `DONE` bit equals 1 and the control register's `I_EN` bit equals 1.

Writing the `status` register clears the `DONE` bit and acknowledges the IRQ. A master peripheral can read the `status` register and determine how the DMA transaction finished by checking the `LEN`, `REOP`, and `WEOP` bits.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 25–6 shows the revision history for this chapter.

Table 25–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	Added a new parameter, **FIFO Depth**.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Updated the Functional Description of the core.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

26. Video Sync Generator and Pixel Converter Cores

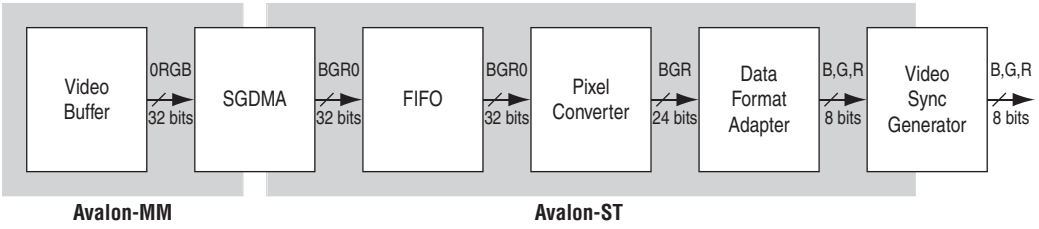
Core Overview

The video sync generator core accepts a continuous stream of pixel data in RGB format, and outputs the data to an off-chip display controller with proper timing. You can configure the video sync generator core to support different display resolutions and synchronization timings.

The pixel converter core transforms the pixel data to the format required by the video sync generator. Figure 26–1 shows a typical placement of the video sync generator and pixel converter cores in a system.

In this example, the video buffer stores the pixel data in 32-bit unpacked format. The extra byte in the pixel data is discarded by the pixel converter core before the data is serialized and sent to the video sync generator core.

Figure 26–1. Typical Placement in a System



The video sync generator and pixel converter cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

These cores are deployed in the Nios II Embedded Software Evaluation Kit (NEEK), which includes an LCD display daughtercard assembly attached via an HSMC connector.

This chapter contains the following sections:

- "Video Sync Generator" on page 26–2
- "Pixel Converter" on page 26–5
- "Device Support" on page 26–6
- "Hardware Simulation Considerations" on page 26–6

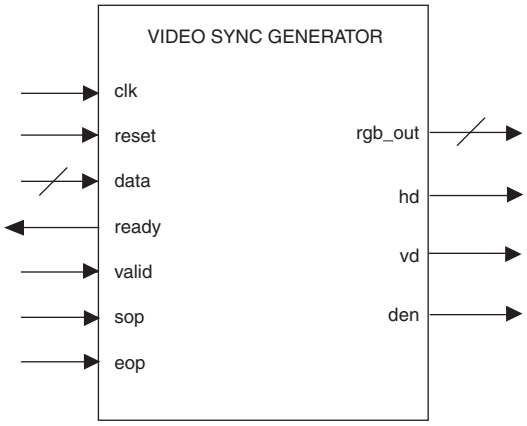
Video Sync Generator

This section describes the hardware structure and functionality of the video sync generator core.

Functional Description

The video sync generator core adds horizontal and vertical synchronization signals to the pixel data that comes through its Avalon® (Avalon-ST) input interface and outputs the data to an off-chip display controller. No processing or validation is performed on the pixel data. Figure 26–2 shows a block diagram of the video sync generator.

Figure 26–2. Video Sync Generator Block Diagram



You can configure various aspects of the core and its Avalon-ST interface to suit your requirements. You can specify the data width, number of beats required to transfer each pixel and synchronization signals. See "Instantiating the Core in SOPC Builder" on page 26–3 for more information on the available options.

To ensure incoming pixel data is sent to the display controller with correct timing, the video sync generator core must synchronize itself to the first pixel in a frame. The first active pixel is indicated by an `sop` pulse.

The video sync generator core expects continuous streams of pixel data at its input interface and assumes that each incoming packet contains the correct number of pixels (Number of rows * Number of columns). Data starvation disrupts synchronization and results in unexpected output on the display.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the video sync generator core in SOPC Builder to configure the core. Table 26–1 lists the available parameters in the MegaWizard interface.

Table 26–1. Video Sync Generator Parameters

Parameter Name	Description
Horizontal Sync Pulse Pixels	The width of the h-sync pulse in number of pixels.
Total Vertical Scan Lines	The total number of lines in one video frame. The value is the sum of the following parameters: **Number of Rows**, **Vertical Blank Lines**, and **Vertical Front Porch Lines**.
Number of Rows	The number of active scan lines in each video frame.
Horizontal Sync Pulse Polarity	The polarity of the h-sync pulse; 0 = active low and 1 = active high.
Horizontal Front Porch Pixels	The number of blanking pixels that follow the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Vertical Sync Pulse Polarity	The polarity of the v-sync pulse; 0 = active low and 1 = active high.
Vertical Sync Pulse Lines	The width of the v-sync pulse in number of lines.
Vertical Front Porch Lines	The number of blanking lines that follow the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Number of Columns	The number of active pixels in each line.
Horizontal Blank Pixels	The number of blanking pixels that precede the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Total Horizontal Scan Pixels	The total number of pixels in one line. The value is the sum of the following parameters: **Number of Columns**, **Horizontal Blank Pixel**, and **Horizontal Front Porch Pixels**.
Beats Per Pixel	The number of beats required to transfer one pixel. Valid values are 1 and 3. This parameter, when multiplied by **Data Stream Bit Width** must be equal to the total number of bits in one pixel. This parameter affects the operating clock frequency, as shown in the following equation: Operating clock frequency = (**Beats per pixel**) * (Pixel_rate), where Pixel_rate (in MHz) = ((**Total Horizontal Scan Pixels**) * (**Total Vertical Scan Lines**) * (Display refresh rate in Hz))/1000000.
Vertical Blank Lines	The number of blanking lines that proceed the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Data Stream Bit Width	The width of the inbound and outbound data.

Signals

Table 26–2 lists the input and output signals for the video sync generator core.

Table 26–2. Video Sync Generator Core Signals

Signal Name	Width (Bits)	Direction	Description
Global Signals			
`clk`	1	Input	System clock.
reset	1	Input	System reset.
Avalon-ST Signals			
`data`	Variable-width	Input	Incoming pixel data. The datawidth is determined by the parameter **Data Stream Bit Width**.
`ready`	1	Output	This signal is asserted when the video sync generator is ready to receive the pixel data.
`valid`	1	Input	This signal is not used by the video sync generator core because the core always expects valid pixel data on the next clock cycle after the `ready` signal is asserted.
`sop`	1	Input	Start-of-packet. This signal is asserted when the first pixel is received.
`eop`	1	Input	End-of-packet. This signal is asserted when the last pixel is received.
LCD Output Signals			
`rgb_out`	Variable-width	Output	Display data. The datawidth is determined by the parameter **Data Stream Bit Width**.
`hd`	1	Output	Horizontal synchronization pulse for display.
`vd`	1	Output	Vertical synchronization pulse for display.
`den`	1	Output	This signal is asserted when the video sync generator core outputs valid data for display.

Timing Diagrams

The horizontal and vertical synchronization timings are determined by the parameters setting. Figure 26–3 shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 8 and 3, respectively.

Figure 26–3. Horizontal Synchronization Timing—8 Bits DataWidth and 3 Beats Per Pixel

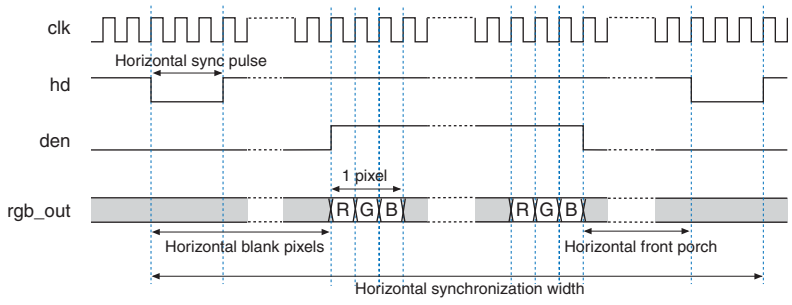


Figure 26–4 sho.ws the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 24 and 1, respectively.

Figure 26–4. Horizontal Synchronization Timing—24 Bits DataWidth and 1 Beat Per Pixel

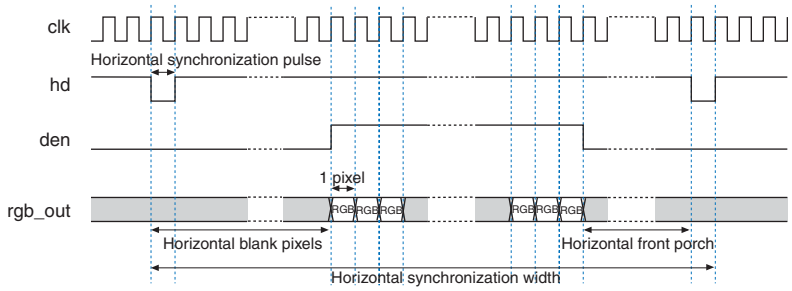
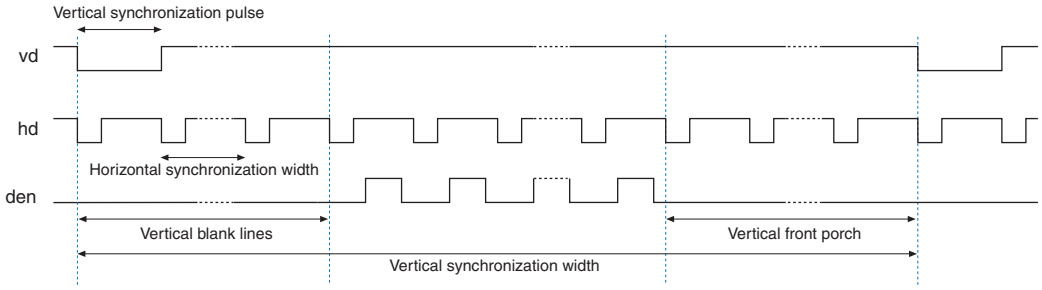


Figure 26–5 shows the vertical synchronization timing.

Figure 26–5. Vertical Synchronization Timing—8 Bits DataWidth and 3 Beats Per Pixel / 24 Bits DataWidth and 1 Beat Per Pixel



Pixel Converter

This section describes the hardware structure and functionality of the pixel converter core.

Functional Description

The pixel converter core receives pixel data on its Avalon-ST input interface and transforms the pixel data to the format required by the video sync generator. The least significant byte of the 32-bit wide pixel data is removed and the remaining 24 bits are wired directly to the core's Avalon-ST output interface.

Instantiating the Core in SOPC Builder

Use the MegaWizard interface for the pixel converter core in SOPC Builder to add the core to a system. You can configure the following parameter:

Source symbols per beat—The number of symbols per beat on the Avalon-ST source interface.

Signals

Table 26–3 lists the input and output signals for the pixel converter core.

Table 26–3. Pixel Converter Input Interface Signals

Signal Name	Width (Bits)	Direction	Description
Global Signals			
clk	1	Input	Not in use.
reset_n	1	Input	
Avalon-ST Signals			
`data_in`	32	Input	Incoming pixel data. Contains four 8-bit symbols that are transferred in 1 beat.
`data_out`	24	Output	Output data. Contains three 8-bit symbols that are transferred in 1 beat.
`sop_in`	1	Input	Wired directly to the corresponding output signals.
`eop_in`	1	Input	
`ready_in`	1	Input	
`valid_in`	1	Input	
`empty_in`	1	Input	
`sop_out`	1	Output	Wired directly from the input signals.
`eop_out`	1	Output	
`ready_out`	1	Output	
`valid_out`	1	Output	
`empty_out`	1	Output	

Device Support

The video sync generator and pixel converter cores support all Altera device families.

Hardware Simulation Considerations

For a typical 60 Hz refresh rate, set the simulation length for the video sync generator core to at least 16.7 µs to get a full video frame. Depending on the size of the video frame, simulation may take a very long time to complete.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 26–4 shows the revision history for this chapter.

Table 26–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Added new parameters for both cores.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

27. Interval Timer Core

Core Overview

The Interval Timer core with Avalon® interface is an interval timer for Avalon-based processor systems, such as a Nios® II processor system. The core provides the following features:

- 32-bit and 64-bit counters.
- Controls to start, stop, and reset the timer.
- Two count modes: count down once and continuous count-down.
- Count-down period register.
- Option to enable or disable the interrupt request (IRQ) when timer reaches zero.
- Optional watchdog timer feature that resets the system if timer ever reaches zero.
- Optional periodic pulse generator feature that outputs a pulse when timer reaches zero.
- Compatible with 32-bit and 16-bit processors.

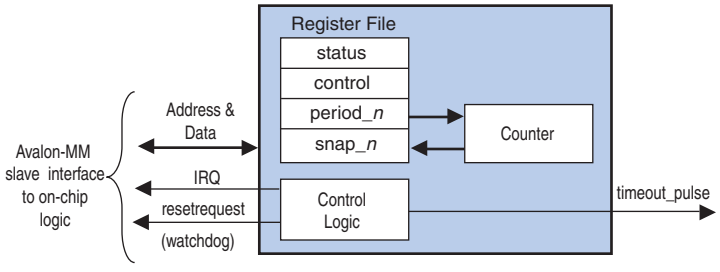
Device drivers are provided in the HAL system library for the Nios II processor. The interval timer core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 27–2
- "Instantiating the Core in SOPC Builder" on page 27–3
- "Software Programming Model" on page 27–5

Functional Description

Figure 27–1 shows a block diagram of the interval timer core.

Figure 27–1. Interval Timer Core Block Diagram



The intervanl timer core has two user-visible features:

- The Avalon Memory-Mapped (Avalon-MM) interface that provides access to six 16-bit registers
- An optional pulse output that can be used as a periodic pulse generator

All registers are 16-bits wide, making the core compatible with both 16-bit and 32-bit processors. Certain registers only exist in hardware for a given configuration. For example, if the core is configured with a fixed period, the period registers do not exist in hardware.

The following sequence describes the basic behavior of the interval timer core:

- An Avalon-MM master peripheral, such as a Nios II processor, writes the core's `control` register to perform the following tasks:
  - Start and stop the timer
  - Enable/disable the IRQ
  - Specify count-down once or continuous count-down mode
- A processor reads the `status` register for information about current timer activity.
- A processor can specify the timer period by writing a value to the period registers.
- An internal counter counts down to zero, and whenever it reaches zero, it is immediately reloaded from the period registers.
- A processor can read the current counter value by first writing to one of the snap registers to request a coherent snapshot of the counter, and then reading the snap registers for the full value.
- When the count reaches zero, one or more of the following events are triggered:
  - If IRQs are enabled, an IRQ is generated.
  - The optional pulse-generator output is asserted for one clock period.
  - The optional watchdog output resets the system.

Avalon-MM Slave Interface

The interval timer core implements a simple Avalon-MM slave interface to provide access to the register file. The Avalon-MM slave port uses the `resetrequest` signal to implement watchdog timer behavior. This signal is a non-maskable reset signal, and it drives the reset input of all Avalon-MM peripherals in the SOPC Builder system. When the `resetrequest` signal is asserted, it forces any processor connected to the system to reboot. For more information, refer to "Configuring the Timer as a Watchdog Timer" on page 27–4.

Device Support

The interval timer core supports all Altera® device families.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the interval timer core in SOPC Builder to specify the hardware features. This section describes the options available in the MegaWizard Interace.

Timeout Period

The **Timeout Period** setting determines the initial value of the period registers. When the **Writeable period** option is on, a processor can change the value of the period by writing to the period registers. When the **Writeable period** option is off, the period is fixed and cannot be updated at runtime. See "Hardware Options" on page 27–3 for information on register options.

The **Timeout Period** is an integer multiple of the **Timer Frequency**. The **Timer Frequency** is fixed at the frequency setting of the system clock associated with the timer. The **Timeout Period** setting can be specified in units of **µs** (microseconds), **ms** (milliseconds), **seconds** , or **clocks** (number of cycles of the system clock associated with the timer). The actual period depends on the frequency of the system clock associated with the timer. If the period is specified in **µs**, **ms**, or **seconds**, the true period will be the smallest number of clock cycles that is greater or equal to the specified **Timeout Period** value. For example, if the associated system clock has a frequency of 30 **ns**, and the specified **Timeout Period** value is 1 **µs**, the true timeout period will be 1.020 microseconds.

Counter Size

The **Counter Size** setting determines the timer's width, which can be set to either 32 or 64 bits. A 32-bit timer has two 16-bit period registers, whereas a 64-bit timer has four 16-bit period registers. This option applies to the snap registers as well.

Hardware Options

The following options affect the hardware structure of the interval timer core. As a convenience, the **Preset Configurations** list offers several pre-defined hardware configurations, such as:

- **Simple periodic interrupt**—This configuration is useful for systems that require only a periodic IRQ generator. The period is fixed and the timer cannot be stopped, but the IRQ can be disabled.
- **Full-featured**—This configuration is useful for embedded processor systems that require a timer with variable period that can be started and stopped under processor control.
- **Watchdog**—This configuration is useful for systems that require watchdog timer to reset the system in the event that the system has stopped responding. Refer to "Configuring the Timer as a Watchdog Timer" on page 27–4.

Register Options

Table 27–1 shows the settings that affect the interval timer core's registers.

Table 27–1. Register Options

Option	Description
Writeable period	When this option is enabled, a master peripheral can change the count-down period by writing to the period registers. When disabled, the count-down period is fixed at the specified **Timeout Period**, and the period registers do not exist in hardware.
Readable snapshot	When this option is enabled, a master peripheral can read a snapshot of the current count-down. When disabled, the status of the counter is detectable only via other indicators, such as the `status` register or the IRQ signal. In this case, the snap registers do not exist in hardware, and reading these registers produces an undefined value.
Start/Stop control bits	When this option is enabled, a master peripheral can start and stop the timer by writing the START and STOP bits in the `control` register. When disabled, the timer runs continuously. When the **System reset on timeout (watchdog)** option is enabled, the START bit is also present, regardless of the **Start/Stop control bits** option.

Output Signal Options

Table 27–2 shows the settings that affect the interval timer core's output signals.

Table 27–2. Output Signal Options

Option	Description
Timeout pulse (1 clock wide)	When this option is on, the core outputs a signal `timeout_pulse`. This signal pulses high for one clock cycle whenever the timer reaches zero. When this option is off, the `timeout_pulse` signal does not exist.
System reset on timeout (watchdog)	When this option is on, the core's Avalon-MM slave port includes the `resetrequest` signal. This signal pulses high for one clock cycle whenever the timer reaches zero resulting in a system-wide reset. The internal timer is stopped at reset. Explicitly writing the START bit of the `control` register starts the timer. When this option is off, the `resetrequest` signal does not exist. Refer to "Configuring the Timer as a Watchdog Timer".

Configuring the Timer as a Watchdog Timer

To configure the core for use as a watchdog, in the MegaWizard Interface select **Watchdog** in the **Preset Configurations** list, or choose the following settings:

- Set the **Timeout Period** to the desired "watchdog" period.
- Turn off **Writeable period**.
- Turn off **Readable snapshot**.
- Turn off **Start/Stop control bits**.
- Turn off **Timeout pulse**.
- Turn on **System reset on timeout (watchdog)**.

A watchdog timer wakes up (comes out of reset) stopped. A processor later starts the timer by writing a 1 to the `control` register's `START` bit. Once started, the timer can never be stopped. If the internal counter ever reaches zero, the watchdog timer resets the system by generating a pulse on its `resetrequest` output. To prevent the system from resetting, the processor must periodically reset the timer's count-down value by writing one of the period registers (the written value is ignored). If the processor fails to access the timer because, for example, software stopped executing normally, the watchdog timer resets the system and returns the system to a defined state.

Software Programming Model

The following sections describe the software programming model for the interval timer core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides hardware abstraction layer (HAL) system library drivers that enable you to access the interval timer core using the HAL application programming interface (API) functions.

HAL System Library Support

The Altera-provided drivers integrate into the HAL system library for Nios II systems. When possible, HAL users should access the core via the HAL API, rather than accessing the core's registers directly.

Altera provides a driver for both the HAL timer device models: system clock timer, and timestamp timer.

System Clock Driver

When configured as the system clock, the interval timer core runs continuously in periodic mode, using the default period set in SOPC builder. The system clock services are then run as a part of the interrupt service routine for this timer. The driver is interrupt-driven, and therefore must have its interrupt signal connected in the system hardware.

The Nios II integrated development environment (IDE) allows you to specify system library properties that determine which timer device will be used as the system clock timer.

Timestamp Driver

The interval timer core may be used as a timestamp device if it meets the following conditions:

- The timer has a writeable `period` register, as configured in SOPC Builder.
- The timer is not selected as the system clock.

The Nios II IDE allows you to specify system library properties that determine which timer device will be used as the timestamp timer.

If the timer hardware is not configured with writeable `period` registers, calls to the `alt_timestamp_start()` API function will not reset the timestamp counter. All other HAL API calls will perform as expected.

For more information about using the system clock and timestamp features that use these drivers, refer to the *Nios II Software Developer's Handbook*. The Nios II Embedded Design Suite (EDS) also provides several example designs that use the interval timer core.

Limitations

The HAL driver for the interval timer core does not support the watchdog reset feature of the core.

Software Files

The interval timer core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_timer_regs.h—**This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_timer.h, altera_avalon_timer_sc.c, altera_avalon_timer_ts.c, altera_avalon_timer_vars.c—**These files implement the timer device drivers for the HAL system library.

Register Map

You do not need to access the interval timer core directly via its registers if using the standard features provided in the HAL system library for the Nios II processor. In general, the register map is only useful to programmers writing a device driver.

CAUTION

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate correctly.

Table 27–3 shows the register map for the 32-bit timer. The interval timer core uses native address alignment. For example, to access the `control` register value, use offset 0x4.

Table 27–3. Register Map—32-bit Timer

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	`status`	RW	*(1)*					`RUN`	`TO`
1	`control`	RW	*(1)*			`STOP`	`START`	`CONT`	`ITO`
2	`periodl`	RW	Timeout Period – 1 (bits [15:0])						
3	`periodh`	RW	Timeout Period – 1 (bits [31:16])						
4	`snapl`	RW	Counter Snapshot (bits [15:0])						
5	`snaph`	RW	Counter Snapshot (bits [31:16])						

Note to Table 27–3:

(1) Reserved. Read values are undefined. Write zero.

For more information about native address alignment, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces*.

Table 27–4 shows the register map for the 64-bit timer.

Table 27–4. Register Map—64-bit Timer

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	*(1)*					RUN	TO
1	control	RW	*(1)*			STOP	START	CONT	ITO
2	period_0	RW	Timeout Period – 1 (bits [15:0])						
3	period_1	RW	Timeout Period – 1 (bits [31:16])						
4	period_2	RW	Timeout Period – 1 (bits [47:32])						
5	period_3	RW	Timeout Period – 1 (bits [63:48])						
6	snap_0	RW	Counter Snapshot (bits [15:0])						
7	snap_1	RW	Counter Snapshot (bits [31:16])						
8	snap_2	RW	Counter Snapshot (bits [47:32])						
9	snap_3	RW	Counter Snapshot (bits [63:48])						

Note to Table 27–4:

(1) Reserved. Read values are undefined. Write zero.

status Register

The `status` register has two defined bits, as shown in Table 27–5.

Table 27–5. status Register Bits

Bit	Name	R/W/C	Description
0	TO	RC	The TO (timeout) bit is set to 1 when the internal counter reaches zero. Once set by a timeout event, the TO bit stays set until explicitly cleared by a master peripheral. Write zero to the status register to clear the TO bit.
1	RUN	R	The RUN bit reads as 1 when the internal counter is running; otherwise this bit reads as 0. The RUN bit is not changed by a write operation to the status register.

control Register

The `control` register has four defined bits, as shown in Table 27–6.

Table 27–6. control Register Bits (Part 1 of 2)

Bit	Name	R/W/C	Description
0	ITO	RW	If the ITO bit is 1, the interval timer core generates an IRQ when the status register's TO bit is 1. When the ITO bit is 0, the timer does not generate IRQs.
1	CONT	RW	The CONT (continuous) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter runs continuously until it is stopped by the STOP bit. If CONT is 0, the counter stops after it reaches zero. When the counter reaches zero, it reloads with the value stored in the period registers, regardless of the CONT bit.
2	START *(1)*	W	Writing a 1 to the START bit starts the internal counter running (counting down). The START bit is an event bit that enables the counter when a write operation is performed. If the timer is stopped, writing a 1 to the START bit causes the timer to restart counting from the number currently stored in its counter. If the timer is already running, writing a 1 to START has no effect. Writing 0 to the START bit has no effect.

Table 27–6. control Register Bits (Part 2 of 2)

Bit	Name	R/W/C	Description
3	STOP *(1)*	W	Writing a 1 to the `STOP` bit stops the internal counter. The `STOP` bit is an event bit that causes the counter to stop when a write operation is performed. If the timer is already stopped, writing a 1 to `STOP` has no effect. Writing a 0 to the stop bit has no effect. If the timer hardware is configured with **Start/Stop control bits** off, writing the `STOP` bit has no effect.

Note to Table 27–6:

(1) Writing 1 to both `START` and `STOP` bits simultaneously produces an undefined result.

period_n Registers

The `period_n` registers together store the timeout period value. The internal counter is loaded with the value stored in these registers whenever one of the following occurs:

- A write operation to one of the `period_n` register
- The internal counter reaches 0

The timer's actual period is one cycle greater than the value stored in the `period_n` registers because the counter assumes the value zero for one clock cycle.

Writing to one of the `period_n` registers stops the internal counter, except when the hardware is configured with **Start/Stop control bits** off. If **Start/Stop control bits** is off, writing either register does not stop the counter. When the hardware is configured with **Writeable period** disabled, writing to one of the `period_n` registers causes the counter to reset to the fixed **Timeout Period** specified at system generation time.

snap_n Registers

A master peripheral may request a coherent snapshot of the current internal counter by performing a write operation (write-data ignored) to one of the `snap_n` registers. When a write occurs, the value of the counter is copied to `snap_n` registers. The snapshot occurs whether or not the counter is running. Requesting a snapshot does not change the internal counter's operation.

Interrupt Behavior

The interval timer core generates an IRQ whenever the internal counter reaches zero and the `ITO` bit of the `control` register is set to 1. Acknowledge the IRQ in one of two ways:

- Clear the `TO` bit of the `status` register
- Disable interrupts by clearing the `ITO` bit of the `control` register

Failure to acknowledge the IRQ produces an undefined result.

Referenced Documents

This chapter references the *Nios II Software Developer's Handbook*.

Document Revision History

Table 27–7 shows the revision history for this chapter.

Table 27–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	Revised descriptions of register fields and bits.	The timer component is using native address alignment.
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Updated the core's name to reflect the name used in SOPC Builder.	—
May 2008 v8.0.0	Added a new parameter and register map for the 64-bit timer.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

Core Overview

Multiprocessor environments can use the mutex core with Avalon® interface to coordinate accesses to a shared resource. The mutex core provides a protocol to ensure mutually exclusive ownership of a shared resource.

The mutex core provides a hardware-based atomic test-and-set operation, allowing software in a multiprocessor environment to determine which processor owns the mutex. The mutex core can be used in conjunction with shared memory to implement additional interprocessor coordination features, such as mailboxes and software mutexes.

The mutex core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera provides device drivers for the Nios II processor to enable use of the hardware mutex.

The mutex core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 28–2
- "Instantiating the Core in SOPC Builder" on page 28–2
- "Software Programming Model" on page 28–2
- "Mutex API" on page 28–4

Functional Description

The mutex core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to two memory-mapped, 32-bit registers. Table 28–1 shows the registers.

Table 28–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 16	15 1	0
0	`mutex`	RW	`OWNER`	`VALUE`	
1	`reset`	RW	Reserved		`RESET`

The mutex core has the following basic behavior. This description assumes there are multiple processors accessing a single mutex core, and each processor has a unique identifier (ID).

- When the `VALUE` field is `0x0000`, the mutex is unlocked and available. Otherwise, the mutex is locked and unavailable.
- The `mutex` register is always readable. Avalon-MM master peripherals, such as a processor, can read the `mutex` register to determine its current state.

- The `mutex` register is writable only under specific conditions. A write operation changes the `mutex` register only if one or both of the following conditions are true:
  - The `VALUE` field of the `mutex` register is zero.
  - The `OWNER` field of the `mutex` register matches the `OWNER` field in the data to be written.
- A processor attempts to acquire the mutex by writing its ID to the `OWNER` field, and writing a non-zero value to the `VALUE` field. The processor then checks if the acquisition succeeded by verifying the `OWNER` field.
- After system reset, the `RESET` bit in the `reset` register is high. Writing a one to this bit clears it.

Device Support

The mutex core supports all Altera device families.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the mutex core in SOPC Builder to specify the core's hardware features. The MegaWizard Interface provides the following options:

- **Initial Value**—the initial contents of the `VALUE` field after reset. If the **Initial Value** setting is non-zero, you must also specify **Initial Owner**.
- **Initial Owner**—the initial contents of the `OWNER` field after reset. When **Initial Owner** is specified, this owner must release the mutex before it can be acquired by another owner.

Software Programming Model

The following sections describe the software programming model for the mutex core. For Nios II processor users, Altera provides routines to access the mutex core hardware. These functions are specific to the mutex core and directly manipulate low-level hardware. The mutex core cannot be accessed via the HAL API or the ANSI C standard library. In Nios II processor systems, a processor locks the mutex by writing the value of its `cpuid` control register to the OWNER field of the `mutex` register.

Software Files

Altera provides the following software files accompanying the mutex core:

- **altera_avalon_mutex_regs.h**—Defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mutex.h**—Defines data structures and functions to access the mutex core hardware.
- **altera_avalon_mutex.c**—Contains the implementations of the functions to access the mutex core

Hardware Access Routines

This section describes the low-level software constructs for manipulating the mutex core. The file **altera_avalon_mutex.h** declares a structure `alt_mutex_dev` that represents an instance of a mutex device. It also declares routines for accessing the mutex hardware structure, listed in Table 28–2.

Table 28–2. Hardware Access Routines

Function Name	Description
`altera_avalon_mutex_open()`	Claims a handle to a mutex, enabling all the other functions to access the mutex core.
`altera_avalon_mutex_trylock()`	Tries to lock the mutex. Returns immediately if it fails to lock the mutex.
`altera_avalon_mutex_lock()`	Locks the mutex. Will not return until it has successfully claimed the mutex.
`altera_avalon_mutex_unlock()`	Unlocks the mutex.
`altera_avalon_mutex_is_mine()`	Determines if this CPU owns the mutex.
`altera_avalon_mutex_first_lock()`	Tests whether the mutex has been released since reset.

These routines coordinate access to the software mutex structure using a hardware mutex core. For a complete description of each function, see section "Mutex API" on page 28–4.

The code shown in Example 28–1 demonstrates opening a mutex device handle and locking a mutex.

Example 28–1. Opening and Locking a mutex

```
#include <altera_avalon_mutex.h>
/* get the mutex device handle */
alt_mutex_dev* mutex = altera_avalon_mutex_open( "/dev/mutex" );
/* acquire the mutex, setting the value to one */
altera_avalon_mutex_lock( mutex, 1 );
/*
 * Access a shared resource here.
 */
/* release the lock */
altera_avalon_mutex_unlock( mutex );
```

Mutex API

This section describes the application programming interface (API) for the mutex core.

altera_avalon_mutex_is_mine()

Prototype: `int altera_avalon_mutex_is_mine(alt_mutex_dev* dev)`

Thread-safe: Yes.

Available from ISR: No.

Include: <**altera_avalon_mutex.h**>

Parameters: `dev`—the mutex device to test.

Returns: Returns non zero if the mutex is owned by this CPU.

Description: `altera_avalon_mutex_is_mine()` determines if this CPU owns the mutex.

altera_avalon_mutex_first_lock()

Prototype: `int altera_avalon_mutex_first_lock(alt_mutex_dev* dev)`

Thread-safe: Yes.

Available from ISR: No.

Include: <**altera_avalon_mutex.h**>

Parameters: `dev`—the mutex device to test.

Returns: Returns 1 if this mutex has not been released since reset, otherwise returns 0.

Description: `altera_avalon_mutex_first_lock()` determines whether this mutex has been released since reset.

altera_avalon_mutex_lock()

Prototype: `void altera_avalon_mutex_lock(alt_mutex_dev* dev, alt_u32 value)`

Thread-safe: Yes.

Available from ISR: No.

Include: <**altera_avalon_mutex.h**>

Parameters: `dev`—the mutex device to acquire.

`value`—the new value to write to the mutex.

Returns: —

Description: `altera_avalon_mutex_lock()` is a blocking routine that acquires a hardware mutex, and at the same time, loads the mutex with the `value` parameter.

altera_avalon_mutex_open()

Prototype:	`alt_mutex_dev* alt_hardware_mutex_open(const char* name)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`name`—the name of the mutex device to open.
Returns:	A pointer to the mutex device structure associated with the supplied name, or NULL if no corresponding mutex device structure was found.
Description:	`altera_avalon_mutex_open()` retrieves a pointer to a hardware mutex device structure.

altera_avalon_mutex_trylock()

Prototype:	`int altera_avalon_mutex_trylock(alt_mutex_dev* dev, alt_u32 value)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to lock. `value`—the new value to write to the mutex.
Returns:	`0` = The mutex was successfully locked. Others = The mutex was not locked.
Description:	`altera_avalon_mutex_trylock()` tries once to lock the hardware mutex, and returns immediately.

altera_avalon_mutex_unlock()

Prototype:	`void altera_avalon_mutex_unlock(alt_mutex_dev* dev)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to unlock.
Returns:	Null.
Description:	`altera_avalon_mutex_unlock()` releases a hardware mutex device. Upon release, the value stored in the mutex is set to zero. If the caller does not hold the mutex, the behavior of this function is undefined.

Document Revision History

Table 28–3 shows the revision history for this chapter.

Table 28–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

29. Mailbox Core

Core Overview

Multiprocessor environments can use the mailbox core with Avalon® interface to send messages between processors.

The mailbox core contains mutexes to ensure that only one processor modifies the mailbox contents at a time. The mailbox core must be used in conjunction with a separate shared memory that is used for storing the actual messages.

The mailbox core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera provides device drivers for the Nios II processor. The mailbox core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 29–2
- "Instantiating the Core in SOPC Builder" on page 29–2
- "Software Programming Model" on page 29–3
- "Mailbox API" on page 29–5

Functional Description

The mailbox core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to four memory-mapped, 32-bit registers. Table 29–1 shows the registers.

Table 29–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 16	15 1	0
0	`mutex0`	RW	OWNER	VALUE	
1	`reset0`	RW	Reserved		RESET
2	`mutex1`	RW	OWNER	VALUE	
3	`reset1`	RW	Reserved		RESET

The mailbox component contains two mutexes: One to ensure unique write access to shared memory and one to ensure unique read access from shared memory. The mailbox core is used in conjunction with a separate memory in the system that is shared among multiple processors.

Mailbox functionality using the mutexes and memory is implemented entirely in the software. Refer to "Software Programming Model" on page 29–3 for details about how to use the mailbox core in software.

For a detailed description of the mutex hardware operation, refer to "Mutex Core" on page 28–1

Device Support

The mailbox core supports all Altera® device families.

Instantiating the Core in SOPC Builder

You can instantiate and configure the mailbox core in an SOPC Builder system using the following process:

1. Decide which processors share the mailbox.
2. On the SOPC Builder **System Contents** tab, instantiate a memory component to serve as the mailbox buffer. Any RAM can be used as the mailbox buffer. The mailbox buffer can share space in an existing memory, such as program memory; it does not require a dedicated memory.
3. On the SOPC Builder **System Contents** tab, instantiate the mailbox component. The mailbox MegaWizard™ Interface presents the following options:
   - **Memory module**—Specifies which memory to use for the mailbox buffer. If the **Memory module** list does not contain the desired shared memory, the memory is not connected in the system correctly. Refer to Step 4 on page 29–2.
   - **CPUs available with this memory**—Shows all the processors that can share the mailbox. This field is always read-only. Use it to verify that the processor connections are correct. If a processor that needs to share the mailbox is missing from the list, refer to Step 4 on page 29–2.
   - **Shared mailbox memory offset**—Specifies an offset into the memory. The mailbox message buffer starts at this offset.
   - **Mailbox size (bytes)**—Specifies the number of bytes to use for the mailbox message buffer. The Nios II driver software provided by Altera uses eight bytes of overhead to implement the mailbox functionality. For a mailbox capable of passing only one message at a time, **Mailbox size (bytes)** must be at least 12 bytes.
   - **Maximum available bytes**—Specifies the number of bytes in the selected memory available for use as the mailbox message buffer. This field is always read-only.
4. If not already connected, make component connections on the SOPC Builder **System Contents** tab.
   a. Connect each processor's data bus master port to the mailbox slave port.
   b. Connect each processor's data bus master port to the shared mailbox memory.

Software Programming Model

The following sections describe the software programming model for the mailbox core. For Nios II processor users, Altera provides routines to access the mailbox core hardware. These functions are specific to the mailbox core and directly manipulate low-level hardware.

The mailbox software programming model has the following characteristics and assumes there are multiple processors accessing a single mailbox core and a shared memory:

- Each mailbox message is one 32-bit word.
- There is a predefined address range in shared memory dedicated to storing messages. The size of this address range imposes a maximum limit on the number of messages pending.
- The mailbox software implements a message FIFO between processors. Only one processor can write to the mailbox at a time, and only one processor can read from the mailbox at a time, ensuring message integrity.
- The software on both the sending and receiving processors must agree on a protocol for interpreting mailbox messages. Typically, processors treat the message as a pointer to a structure in shared memory.
- The sending processor can post messages in succession, up to the limit imposed by the size of the message address range.
- When messages exist in the mailbox, the receiving processor can read messages. The receiving processor can block until a message appears, or it can poll the mailbox for new messages.
- Reading the message removes the message from the mailbox.

Software Files

Altera provides the following software files accompanying the mailbox core hardware:

- **altera_avalon_mailbox_regs.h**—Defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mailbox.h**—Defines data structures and functions to access the mailbox core hardware.
- **altera_avalon_mailbox.c**—Contains the implementations of the functions to access the mailbox core.

Programming with the Mailbox Core

This section describes the software constructs for manipulating the mailbox core hardware.

The file **altera_avalon_mailbox.h** declares a structure `alt_mailbox_dev` that represents an instance of a mailbox device. It also declares functions for accessing the mailbox hardware structure, listed in Table 29–2. For a complete description of each function, refer to "Mailbox API" on page 29–5.

Table 29–2. Mailbox API Functions

Function Name	Description
`altera_avalon_mailbox_close()`	Closes the handle to a mailbox.
`altera_avalon_mailbox_get()`	Returns a message if one is present, but does not block waiting for a message.
`altera_avalon_mailbox_open()`	Claims a handle to a mailbox, enabling all the other functions to access the mailbox core.
`altera_avalon_mailbox_pend()`	Blocks waiting for a message to be in the mailbox.
`altera_avalon_mailbox_post()`	Posts a message to the mailbox.

Example 29–1 demonstrates writing to and reading from a mailbox. For this example, assume that the hardware system has two processors communicating via mailboxes. The system includes two mailbox cores, which provide two-way communication between the processors.

Example 29–1. Writing to and Reading from a Mailbox

```
#include <stdio.h>
#include "altera_avalon_mailbox.h"

int main()
{
  alt_u32 message = 0;
  alt_mailbox_dev* send_dev, recv_dev;
  /* Open the two mailboxes between this processor and another */
  send_dev = altera_avalon_mailbox_open("/dev/mailbox_0");
  recv_dev = altera_avalon_mailbox_open("/dev/mailbox_1");

  while(1)
  {
    /* Send a message to the other processor */
    altera_avalon_mailbox_post(send_dev, message);

    /* Wait for the other processor to send a message back */
    message = altera_avalon_mailbox_pend(recv_dev);
  }
  return 0;
}
```

Mailbox API

This section describes the application programming interface (API) for the mailbox core.

altera_avalon_mailbox_close()

Prototype:	`void altera_avalon_mailbox_close (alt_mailbox_dev* dev);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox to close.
Returns:	Null.
Description:	`altera_avalon_mailbox_close()` closes the mailbox.

altera_avalon_mailbox_get()

Prototype:	`alt_u32 altera_avalon_mailbox_get (alt_mailbox_dev* dev, int* err);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox handle. `err`—pointer to an error code that is returned.
Returns:	Returns a message if one is available in the mailbox, otherwise returns 0. The value pointed to by `err` is 0 if the message was read correctly, or `EWOULDBLOCK` if there is no message to read.
Description:	`altera_avalon_mailbox_get()` returns a message if one is present, but does not block waiting for a message.

altera_avalon_mailbox_open()

Prototype:	`alt_mailbox_dev* altera_avalon_mailbox_open (const char* name);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`name`—the name of the mailbox device to open.
Returns:	Returns a handle to the mailbox, or NULL if this mailbox does not exist.
Description:	`altera_avalon_mailbox_open()` opens a mailbox.

altera_avalon_mailbox_pend()

Prototype: `alt_u32 altera_avalon_mailbox_pend (alt_mailbox_dev* dev);`

Thread-safe: Yes.

Available from ISR: No.

Include: <**altera_avalon_mailbox.h**>

Parameters: `dev`—the mailbox device to read a message from.

Returns: Returns the message.

Description: `altera_avalon_mailbox_pend()` is a blocking routine that waits for a message to appear in the mailbox and then reads it.

altera_avalon_mailbox_post()

Prototype: `int altera_avalon_mailbox_post (alt_mailbox_dev* dev, alt_u32 msg);`

Thread-safe: Yes.

Available from ISR: No.

Include: <**altera_avalon_mailbox.h**>

Parameters: `dev`—the mailbox device to post a message to.
`msg`—the value to post.

Returns: Returns 0 on success, or `EWOULDBLOCK` if the mailbox is full.

Description: `altera_avalon_mailbox_post()` posts a message to the mailbox.

Document Revision History

Table 29–3 shows the revision history for this chapter.

Table 29–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

30. Vectored Interrupt Controller Core

Core Overview

The vectored interrupt controller (VIC) core serves the following main purposes:

- Provides an interface to the interrupts in your system
- Reduces interrupt overhead
- Manages large numbers of interrupts

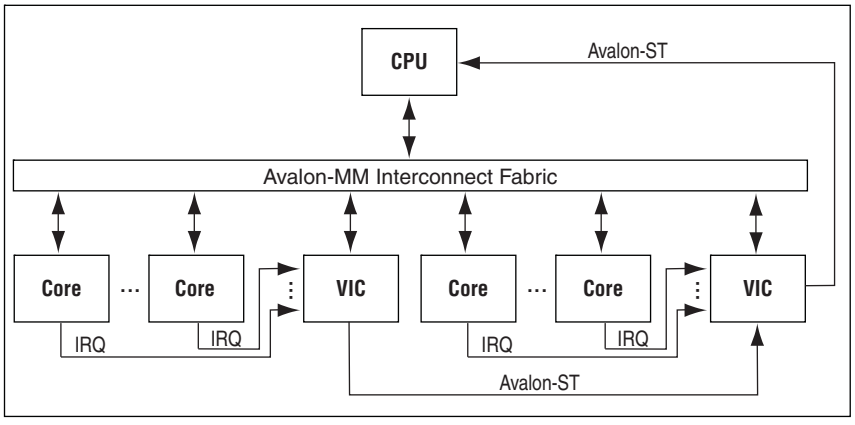
The VIC offers high-performance, low-latency interrupt handling. The VIC prioritizes interrupts in hardware and outputs information about the highest-priority pending interrupt. When external interrupts occur in a system containing a VIC, the VIC determines the highest priority interrupt, determines the source that is requesting service, computes the requested handler address (RHA), and provides information, including the RHA, to the processor.

The VIC core contains the following interfaces:

- Up to 32 interrupt input ports per VIC core
- One Avalon® Memory-Mapped (Avalon-MM) slave interface to access the internal control status registers (CSR)
- One Avalon Streaming (Avalon-ST) interface output interface to pass information about the selected interrupt
- One optional Avalon-ST interface input interface to receive the Avalon-ST output in systems with daisy-chained VICs

Figure 30–1 outlines the basic layout of a system containing two VIC components.

Figure 30–1. Sample System Layout



To use the VIC, the processor in your system needs to have a matching Avalon-ST interface to accept the interrupt information, such as the Nios® II processor's external interrupt controller interface.

The characteristics of each interrupt port are configured via the Avalon-MM slave interface. When you need more than 32 interrupt ports, you can daisy chain multiple VICs together.

The VIC core provides the following features:

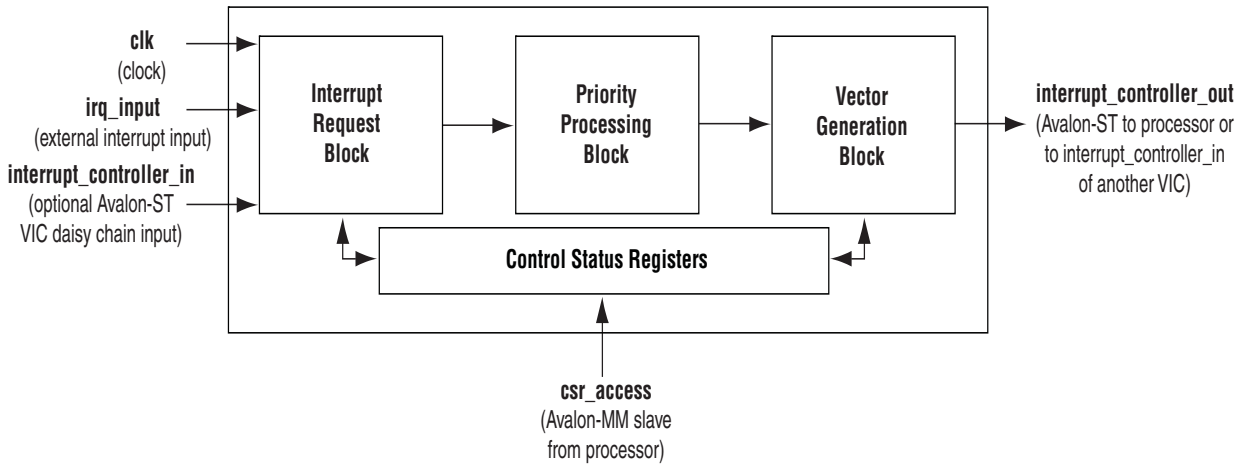
- Separate programmable requested interrupt level (RIL) for each interrupt
- Separate programmable requested register set (RRS) for each interrupt, to tell the interrupt handler which processor register set to use
- Separate programmable requested non-maskable interrupt (RNMI) flag for each interrupt, to control whether each interrupt is maskable or non-maskable
- Software-controlled priority arbitration scheme

The VIC core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios II processor, Altera provides Hardware Abstraction Layer (HAL) driver routines for the VIC core. Refer to "Altera HAL Software Programming Model" on page 30–10 for HAL support details.

Functional Description

Figure 30–2 shows a high-level block diagram of the VIC core.

Figure 30–2. VIC Block Diagram



External Interfaces

The following sections describe the external interfaces for the VIC core.

clk

`clk` is a system clock interface. This interface connects to your system's main clock source. The interface's signals are `clk` and `reset_n`.

irq_input

`irq_input` comprises up to 32 single-bit, level-sensitive Avalon interrupt receiver interfaces. These interfaces connect to interrupt sources. There is one `irq` signal for each interface.

interrupt_controller_out

`interrupt_controller_out` is an Avalon-ST output interface, as defined in Table 30–2, configured with a ready latency of 0 cycles. This interface connects to your processor or to the `interrupt_controller_in` interface of another VIC. The interface's signals are `valid` and `data`. Table 30–1 shows the interface's parameters and the corresponding parameter values.

Table 30–1. interrupt_controller_out and interrupt_controller_in Parameters

Parameter	Value
Symbol width	45 bits
Ready latency	0 cycles

interrupt_controller_in

`interrupt_controller_in` is an optional Avalon-ST input interface, as defined in Table 30–2, configured with a ready latency of 0 cycles. Include this interface in the second, third, etc, VIC components of a daisy-chained multiple VIC system. This interface connects to the `interrupt_controller_out` interface of the immediately-preceding VIC in the chain. The interface's signals are `valid` and `data`. Table 30–1 shows the interface's parameters and the corresponding parameter values.

The `interrupt_controller_out` and `interrupt_controller_in` interfaces have identical Avalon-ST formats so you can daisy chain VICs together in SOPC Builder when you need more than 32 interrupts. `interrupt_controller_out` always provides valid data and cannot be back-pressured. Table 30–2 shows the fields of the VIC's 45-bit Avalon-ST interface.

Table 30–2. VIC Avalon-ST Interface Fields

44 43 42 41 40 39 38 38 37 ... 20 19 18 17 16 15 14 13	12 11 10 9 8 7	6	5 4 3 2 1 0
RHA *(1)*	RRS *(2)*	RNMI *(2)*	RIL *(2)*

Notes to Table 30–2:

(1) RHA contains the 32-bit address of the interrupt handling routine.

(2) Refer to Table 30–6 for a description of this field.

csr_access

`csr_access` is a VIC CSR interface consisting of an Avalon-MM slave interface. This interface connects to the data master of your processor. The interface's signals are `read`, `write`, `address`, `readdata`, and `writedata`. Table 30–3 shows the interface's parameters and the corresponding parameter values.

Table 30–3. csr_access Parameters

Parameter	Value
Read wait	1 cycle
Write wait	0 cycles
Ready latency	4 cycles

For information about the Avalon-MM slave and Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Functional Blocks

The following main design blocks comprise the VIC core:

- Interrupt request block
- Priority processing block
- Vector generation block

The following sections describe each functional block.

Interrupt Request Block

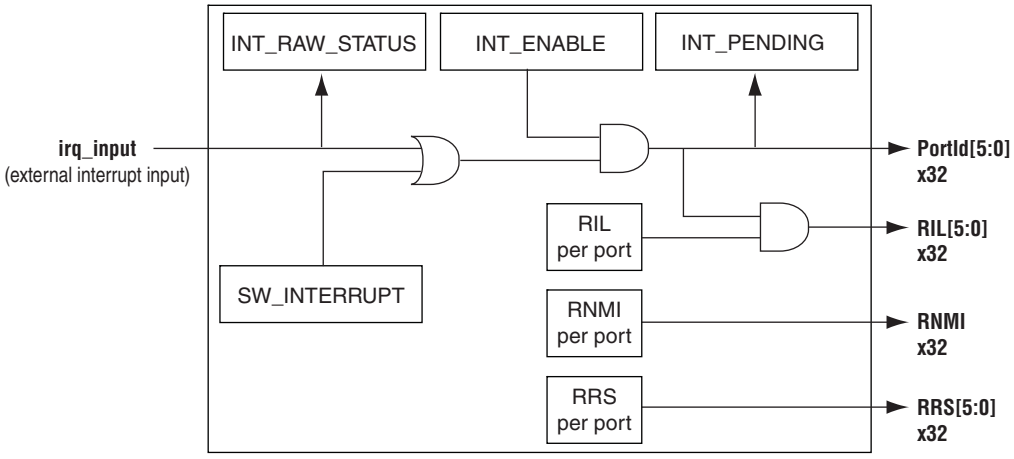
The interrupt request block controls the input interrupts, providing functionality such as setting interrupt levels, setting the per-interrupt programmable registers, masking interrupts, and managing software-controlled interrupts. You configure the number of interrupt input ports when you create the component. Refer to "Instantiating the Core in SOPC Builder" on page 30–9 for configuration options.

This block contains the majority of the VIC CSRs. The CSRs are accessed via the Avalon-MM slave interface.

Optional output from another VIC core can also come into the interrupt request block. Refer to "Daisy Chaining VIC Cores" on page 30–5 for more information.

Figure 30–3 shows the details of the interrupt request block. Each interrupt can be driven either by its associated `irq_input` signal (connected to a component with an interrupt source) or by a software trigger controlled by a CSR (even when there is no interrupt source connected to the `irq_input` signal).

Figure 30–3. Interrupt Request Block



Priority Processing Block

The priority processing block chooses the interrupt with the highest priority. The block receives information for each interrupt from the interrupt request block and passes information for the highest priority interrupt to the vector generation block.

The interrupt request with the numerically-largest RIL has priority. If multiple interrupts are pending with the same numerically-largest RIL, the numerically-lowest IRQ index of those interrupts has priority.

The RIL is a programmable interrupt level per port. An RIL value of zero disables the interrupt. You configure the bit width of the RIL when you create the component. Refer to "Instantiating the Core in SOPC Builder" on page 30–9 for configuration options.

Vector Generation Block

The vector generation block receives information for the highest priority interrupt from the priority processing block. The vector generation block uses the port identifier passed from the priority processing block along with the vector base address and bytes per vector programmed in the CSRs during software initialization to compute the RHA. Equation 30–1 shows the RHA formula.

Equation 30–1. RHA Calculation

$$RHA = (\text{port identifier} \times \text{bytes per vector}) + \text{vector base address}$$

The information then passes out of the vector generation block and the VIC using the Avalon-ST interface. Refer to Table 30–2 on page 30–3 for details about the outgoing information. The output from the VIC typically connects to a processor or another VIC, depending on the design.

Daisy Chaining VIC Cores

You can create a system with more than 32 interrupts by daisy chaining multiple VIC cores together. This is done by connecting the `interrupt_controller_out` interface of one VIC to the optional `interrupt_controller_in` interface of another VIC. For information about enabling the optional input interface, refer to "Instantiating the Core in SOPC Builder" on page 30–9.

☞ For performance reasons, always directly connect VIC components. Do not include other components between VICs.

When daisy chain input comes into the VIC, the priority processing block considers the daisy chain input along with the hardware and software interrupt inputs from the interrupt request block to determine the highest priority interrupt. If the daisy chain input has the highest RIL value, then the vector generation block passes the daisy chain port values unchanged directly out of the VIC.

You can daisy chain VICs with fewer than 32 interrupt ports. The number of daisy chain connections is only limited to the hardware and software resources. Refer to "Latency Information" for details about the impact of multiple VICs.

☞ Altera recommends setting the RIL width to the same value in all daisy-chained VIC components. If your RIL widths are different, wider RILs from upstream VICs are truncated.

Latency Information

The latency of an interrupt request traveling through the VIC is the sum of the delay through each of the blocks. Clock delays in the interrupt request block and the vector generation block are constants. The clock delay in the priority processing block varies depending on the total number of interrupt ports. Table 30–4 shows the latency information.

Table 30–4. Clock Delay Latencies

Number of Interrupt Ports	Interrupt Request Block Delay	Priority Processing Block Delay	Vector Generation Block Delay	Total Interrupt Latency
2 – 4	2 cycles	1 cycle	1 cycle	4 cycles
5 – 16	2 cycles	2 cycles	1 cycle	5 cycles
17 – 32	2 cycles	3 cycles	1 cycle	6 cycles

When daisy-chaining multiple VICs, interrupt latency increases as you move through the daisy chain away from the processor. For best performance, assign interrupts with the lowest latency requirements to the VIC connected directly to the processor.

Register Maps

The VIC core CSRs are accessible through the Avalon-MM interface. Software can configure the core and determine current status by accessing the registers.

☞ Each register has a 32-bit interface that is not byte-enabled. You must access these registers with a master that is at least 32 bits wide.

Table 30–5 lists and describes the registers.

Table 30–5. Control Status Registers (Part 1 of 3)

Offset	Register Name	Access	Reset Value	Description
0 – 31	`INT_CONFIG<n>`	R/W	0	There are 32 interrupt configuration registers (`INT_CONFIG0` – `INT_CONFIG31`). Each register contains fields to configure the behavior of its corresponding interrupt. If an interrupt input does not exist, reading the corresponding register always returns zero, and writing is ignored. Refer to Table 30–6 on page 30–8 for the `INT_CONFIG` register map.
32	`INT_ENABLE`	R/W	0	The interrupt enable register. `INT_ENABLE` holds the enabled status of each interrupt input. The 32 bits of the register map to the 32 interrupts available in the VIC core. For example, bit 5 corresponds to IRQ5. *(1)* Interrupt that are not enabled are never considered by the priority processing block, even when the interrupt input is asserted.
33	`INT_ENABLE_SET`	W	0	The interrupt enable set register. Writing a 1 to a bit in `INT_ENABLE_SET` sets the corresponding bit in `INT_ENABLE`. Writing a 0 to a bit has no effect. Reading from this register always returns 0. *(1)*

Table 30–5. Control Status Registers (Part 2 of 3)

Offset	Register Name	Access	Reset Value	Description
34	INT_ENABLE_CLR	W	0	The interrupt enable clear register. Writing a 1 to a bit in INT_ENABLE_CLR clears corresponding bit in INT_ENABLE. Writing a 0 to a bit has no effect. Reading from this register always returns 0. *(1)*
35	INT_PENDING	R	0	The interrupt pending register. INT_PENDING shows the pending interrupts. Each bit corresponds to one interrupt input. If an interrupt does not exist, reading its corresponding INT_PENDING bit always returns 0, and writing is ignored. Bits in INT_PENDING are set in the following ways: ■ An external interrupt is asserted at the VIC interface and the corresponding INT_ENABLE bit is set. ■ An SW_INTERRUPT bit is set and the corresponding INT_ENABLE bit is set. INT_PENDING bits remain set as long as either condition applies. Refer to Figure 30–3 on page 30–4 for details. *(1)*
36	INT_RAW_STATUS	R	0	The interrupt raw status register. INT_RAW_STATUS shows the unmasked state of the interrupt inputs. If an interrupt does not exist, reading the corresponding INT_RAW_STATUS bit always returns 0, and writing is ignored. A set bit indicates an interrupt is asserted at the interface of the VIC. The interrupt is asserted to the processor only when the corresponding bit in the interrupt enable register is set. *(1)*
37	SW_INTERRUPT	R/W	0	The software interrupt register. SW_INTERRUPT drives the software interrupts. Each interrupt is ORed with its external hardware interrupt and then enabled with INT_ENABLE. Refer to Figure 30–3 on page 30–4 for details. *(1)*
38	SW_INTERRUPT_SET	W	0	The software interrupt set register. Writing a 1 to a bit in SW_INTERRUPT_SET sets the corresponding bit in SW_INTERRUPT. Writing a 0 to a bit has no effect. Reading from this register always returns 0. *(1)*
39	SW_INTERRUPT_CLR	W	0	The software interrupt clear register. Writing a 1 to a bit in SW_INTERRUPT_CLR clears the corresponding bit in SW_INTERRUPT. Writing a 0 to a bit has no effect. Reading from this register always returns 0. *(1)*
40	VIC_CONFIG	R/W	0	The VIC configuration register. VIC_CONFIG allows software to configure settings that apply to the entire VIC. Refer to Table 30–7 on page 30–9 for the VIC_CONFIG register map.
41	VIC_STATUS	R	0	The VIC status register. VIC_STATUS shows the current status of the VIC. Refer to Table 30–8 on page 30–9 for the VIC_STATUS register map.
42	VEC_TBL_BASE	R/W	0	The vector table base register. VEC_TBL_BASE holds the base address of the vector table in the processor's memory space. Because the table must be aligned on a 4-byte boundary, bits 1:0 must always be 0.

Table 30–5. Control Status Registers (Part 3 of 3)

Offset	Register Name	Access	Reset Value	Description
43	VEC_TBL_ADDR	R	0	The vector table address register. VEC_TBL_ADDR provides the RHA for the IRQ value with the highest priority pending interrupt. If no interrupt is active, the value in this register is 0. If daisy chain input is enabled and is the highest priority interrupt, the vector table address register contains the RHA value from the daisy chain input interface.

Note to Table 30–5:

(1) This register contains a 1-bit field for each of the 32 interrupt inputs. When the VIC is configured for less than 32 interrupts, the corresponding 1-bit field for each unused interrupts is tied to zero. Reading these locations always returns 0, and writing is ignored. To determine which interrupts are present, write the value 0xffffffff to the register and then read the register contents. Any bits that return zero do not have an interrupt present.

Table 30–6 provides a bit map for the 32 INT_CONFIG registers.

Table 30–6. The INT_CONFIG Register Map

Bits	Field Name	Access	Reset Value	Description
0:5	RIL	R/W	0	The requested interrupt level field. RIL contains the interrupt level of the interrupt requesting service. The processor can use the value in this field to determine if the interrupt is of higher priority than what the processor is currently doing.
6	RNMI	R/W	0	The requested non-maskable interrupt field. RNMI contains the non-maskable interrupt mode of the interrupt requesting service. When 0, the interrupt is maskable. When 1, the interrupt is non-maskable.
7:12	RRS	R/W	0	The requested register set field. RRS contains the number of the processor register set that the processor should use for processing the interrupt. Software must ensure that only register values supported by the processor are used.
13:31	Reserved			

For expanded definitions of the terms in Table 30–6, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Table 30–7 provides a bit map for the VIC_CONFIG register.

Table 30–7. The VIC_CONFIG Register Map

Bits	Field Name	Access	Reset Value	Description
0:2	VEC_SIZE	R/W	0	The vector size field. VEC_SIZE specifies the number of bytes in each vector table entry. VEC_SIZE is encoded as log2 (number of words) - 2. Namely: ■ 0—4 bytes per vector table entry ■ 1—8 bytes per vector table entry ■ 2—16 bytes per vector table entry ■ 3—32 bytes per vector table entry ■ 4—64 bytes per vector table entry ■ 5—128 bytes per vector table entry ■ 6—256 bytes per vector table entry ■ 7—512 bytes per vector table entry
3	DC	R/W	0	The daisy chain field. DC serves the following purposes: ■ Enables and disables the daisy chain input interface, if present. Write a 1 to enable the daisy chain interface; write a 0 to disable it. ■ Detects the presence of the daisy chain input interface. To detect, write a 1 to DC and then read DC. A return value of 1 means the daisy chain interface is present; 0 means the daisy chain interface is not present.
4:31	Reserved			

Table 30–8 provides a bit map for the VIC_STATUS register.

Table 30–8. The VIC_STATUS Register Map

Bits	Field Name	Access	Reset Value	Description
0:5	HI_PRI_IRQ	R	0	The highest priority interrupt field. HI_PRI_IRQ contains the IRQ number of the active interrupt with the highest RIL. When there is no active interrupt (IP is 0), reading from this field returns 0. When the daisy chain input is enabled and it is the highest priority interrupt, then the value read from this field is 32. Bit 5 always reads back 0 when the daisy chain input is not present.
6:30	Reserved			
31	IP	R	0	The interrupt pending field. IP indicates when there is an interrupt ready to be serviced. A 1 indicates an interrupt is pending; a 0 indicates no interrupt is pending.

Device Support

The VIC core supports all Altera device families currently supported by the Quartus® II software.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the VIC core in SOPC Builder to add the core to a system.

Generation-time parameters control the features present in the hardware. Table 30–9 lists and describes the parameters you can configure.

Table 30–9. Parameters for VIC Core

Parameter	Legal Values	Description
Number of interrupts	1 – 32	Specifies the number of `irq_input` interrupt interfaces.
RIL width	1 – 6	Specifies the bit width of the requested interrupt level.
Daisy chain enable	True / False	Specifies whether or not to include an input interface for daisy chaining VICs together.

Because multiple VICs can exist in a single system, SOPC Builder assigns a unique interrupt controller identification number to each VIC generated.

Keep the following considerations in mind when connecting the core in your SOPC Builder system:

- The CSR access interface (`csr_access`) connects to a data master port on your processor.
- The daisy chain input interface (`interrupt_controller_in`) is only visible when the daisy chain enable option is on.
- The interrupt controller output interface (`interrupt_controller_out`) connects either to the EIC port of your processor, or to another VIC's daisy chain input interface (`interrupt_controller_in`).
- For SOPC Builder interoperability, the VIC core includes an Avalon-MM master port. This master interface is not used to access memory or peripherals. Its purpose is to allow peripheral interrupts to connect to the VIC in SOPC Builder. The port must be connected to an Avalon-MM slave to create a valid SOPC Builder system. Then at system generation time, the unused master port is removed during optimization. The most simple solution is to connect the master port directly into the CSR access interface (`csr_access`).
- SOPC Builder automatically connects interrupt sources when instantiating components. When using the provided HAL device driver for the VIC, daisy chaining multiple VICs in a system requires that each interrupt source is connected to exactly one VIC. You need to manually remove any extra connections.

Altera HAL Software Programming Model

The Altera-provided driver implements a HAL device driver that integrates with a HAL board support package (BSP) for Nios II systems. HAL users should access the VIC core via the familiar HAL API.

Software Files

The VIC driver includes the following software files. These files provide low-level access to the hardware and drivers that integrate with the Nios II HAL BSP. Application developers should not modify these files.

- **altera_vic_regs.h**—Defines the core's register map, providing symbolic constants to access the low-level hardware.

- **altera_vic_funnel.h**, **altera_vic_irq.h**, **altera_vic_irq.h**, **altera_vic_irq_init.h**—Define the prototypes and macros necessary for the VIC driver.
- **altera_vic.c**, **altera_vic_irq_init.c**, **altera_vic_isr_register.c**, **altera_vic_sw_intr.c**, **altera_vic_set_level.c**, **altera_vic_funnel_non_preemptive_nmi.S**, **altera_vic_funnel_non_preemptive.S**, and **altera_vic_funnel_preemptive.S**—Provide the code that implements the VIC driver.
- **altera_**<*name*>**_vector_tbl.S**—Provides a vector table file for each VIC in the system. The BSP generator creates these files.

Macros

Macros to access all of the registers are defined in **altera_vic_regs.h.** For example, this file includes macros to access the `INT_CONFIG` register, including the following macros:

```
#define IOADDR_ALTERA_VIC_INT_CONFIG(base, irq)
      __IO_CALC_ADDRESS_NATIVE(base, irq)
#define IORD_ALTERA_VIC_INT_CONFIG(base, irq)         IORD(base, irq)
#define IOWR_ALTERA_VIC_INT_CONFIG(base, irq, data)   IOWR(base, irq,
data)
#define ALTERA_VIC_INT_CONFIG_RIL_MSK (0x3f)
#define ALTERA_VIC_INT_CONFIG_RIL_OFST (0)
#define ALTERA_VIC_INT_CONFIG_RNMI_MSK (0x40)
#define ALTERA_VIC_INT_CONFIG_RNMI_OFST (6)
#define ALTERA_VIC_INT_CONFIG_RRS_MSK (0x1f80)
#define ALTERA_VIC_INT_CONFIG_RRS_OFST (7)
```

For a complete list of predefined macros and utilities to access the VIC hardware, refer to the following files:

- <*install_dir*>**\ip\altera\altera_vectored_interrupt_controller\top\inc\altera_vic_regs.h**
- <*install_dir*>**\ip\altera\altera_vectored_interrupt_controller\top\HAL\inc\altera_vic_funnel.h**
- <*install_dir*>**\ip\altera\altera_vectored_interrupt_controller\top\HAL\inc\altera_vic_irq.h**

Data Structure

Figure 30–4 shows the data structure for the device.

Figure 30–4. Device Data Structure

```
#define ALT_VIC_MAX_INTR_PORTS          (32)

typedef struct alt_vic_dev
{
  void          *base;                  /* Base address of VIC */
  alt_u32       intr_controller_id;     /* Interrupt controller ID */
  alt_u32       num_of_intr_ports;      /* Number of interrupt ports */
  alt_u32       ril_width;              /* RIL width */
  alt_u32       daisy_chain_present;    /* Daisy-chain input present */
  alt_u32       vec_size;               /* Vector size */
  void          *vec_addr;              /* Vector table base address */
  alt_u32       int_config[ALT_VIC_MAX_INTR_PORTS];   /* INT_CONFIG settings
                                                          for each interrupt */
} alt_vic_dev;
```

VIC API

The VIC device driver provides all the routines required of an Altera HAL external interrupt controller (EIC) device driver. The following functions are required by the Altera Nios II enhanced HAL interrupt API:

- alt_ic_isr_register ()
- alt_ic_irq_enable()
- alt_ic_irq_disable()
- alt_ic_irq_enabled()

These functions write to the register map to change the setting or read from the register map to check the status of the VIC component thru a memory-mapped address.

For detailed descriptions of these functions, refer to the to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Table 30–10 lists the API functions specific to the VIC core and briefly describes each. Details of each function follow the table.

Table 30–10. Function List

Name	Description
`alt_vic_sw_interrupt_set()`	Sets the corresponding bit in the `SW_INTERRUPT` register to enable a given interrupt via software.
`alt_vic_sw_interrupt_clear()`	Clears the corresponding bit in the `SW_INTERRUPT` register to disable a given interrupt via software.
`alt_vic_sw_interrupt_status()`	Reads the status of the `SW_INTERRUPT` register for a given interrupt.
`alt_vic_irq_set_level()`	Sets the interrupt level for a given interrupt.

alt_vic_sw_interrupt_set()

Prototype: int alt_vic_sw_interrupt_set(alt_u32 ic_id, alt_u32 irq)

Thread-safe: No

Available from ISR: No

Include: **altera_vic_irq.h**, **altera_vic_regs.h**

Parameters: `ic_id`—the interrupt controller identification number as defined in **system.h**

`irq`—the interrupt value as defined in **system.h**

Returns: Returns zero if successful; otherwise non-zero for one or more of the following reasons:

- The value in `ic_id` is invalid
- The value in `irq` is invalid

Description: Triggers a single software interrupt

alt_vic_sw_interrupt_clear()

Prototype: int alt_vic_sw_interrupt_clear(alt_u32 ic_id, alt_u32 irq)

Thread-safe: No

Available from ISR: Yes; if interrupt preemption is enabled, disable global interrupts before calling this routine.

Include: **altera_vic_irq.h**, **altera_vic_regs.h**

Parameters: `ic_id`—the interrupt controller identification number as defined in **system.h**

`irq`—the interrupt value as defined in **system.h**

Returns: Returns zero if successful; otherwise non-zero for one or more of the following reasons:

- The value in `ic_id` is invalid
- The value in `irq` is invalid

Description: Clears a single software interrupt

alt_vic_sw_interrupt_status()

Prototype: alt_u32 alt_vic_sw_interrupt_status(alt_u32 ic_id, alt_u32 irq)

Thread-safe: No

Available from ISR: Yes; if interrupt preemption is enabled, disable global interrupts before calling this routine.

Include: **altera_vic_irq.h**, **altera_vic_regs.h**

Parameters: `ic_id`—the interrupt controller identification number as defined in **system.h**

`irq`—the interrupt value as defined in **system.h**

Returns: Returns non-zero if the corresponding software trigger interrupt is active; otherwise zero for one or more of the following reasons:

- The corresponding software trigger interrupt is disabled
- The value in `ic_id` is invalid
- The value in `irq` is invalid

Description: Checks the software interrupt status for a single interrupt

alt_vic_irq_set_level()

Prototype: int alt_vic_irq_set_level(alt_u32 ic_id, alt_u32 irq, alt_u32 level)

Thread-safe: No

Available from ISR: No

Include: **altera_vic_irq.h**, **altera_vic_regs.h**

Parameters: `ic_id`—the interrupt controller identification number as defined in **system.h**

`irq`—the interrupt value as defined in **system.h**

`level`—the interrupt level to set

Returns: Returns zero if successful; otherwise non-zero for one or more of the following reasons:

- The value in `ic_id` is invalid
- The value in `irq` is invalid
- The value in `level` is invalid

Description: Sets the interrupt level for a single interrupt.

Altera recommends setting the interrupt level only to zero to disable the interrupt or to the original value specified in your BSP. Writing any other value could violate the overlapping register set, priority level, and other design rules. Refer to "VIC BSP Design Rules for Altera Hal Implementation" on page 30–18 for more information.

Run-time Initialization

During system initialization, software configures the each VIC instance's control registers using settings specified in the BSP. The RIL, RRS, and RNMI fields are written into the interrupt configuration register of each interrupt port in each VIC. All interrupts are disabled until other software registers a handler using the alt_ic_isr_register() API.

Board Support Package

The BSP you generate for your Nios II system provides access to the hardware in your system, including the VIC. The VIC driver includes scripts that the BSP generator calls to get default interrupt settings and to validate settings during BSP generation. The Nios II BSP Editor provides a mechanism to edit these settings and generate a BSP for your SOPC Builder design.

The generator produces a vector table file for each VIC in the system, named **altera_**<*name*>**_vector_tbl.S**. The vector table's source path is added to the BSP Makefile for compilation along with other VIC driver source code. Its contents are based on the BSP settings for each VIC's interrupt ports.

VIC BSP Settings

The VIC driver scripts provide settings to the BSP. The number and naming of these settings depends on your hardware system's configuration, specifically, the number of optional shadow register sets in the Nios II processor, the number of VIC controllers in the system, and the number of interrupt ports each VIC has.

Certain settings apply to all VIC instances in the system, while others apply to a specific VIC instance. Settings that apply to each interrupt port apply only to the specified interrupt port number on that VIC instance.

The remainder of this section lists details and descriptions of each VIC BSP setting.

altera_vic_driver.enable_preemption

Identifier: ALTERA_VIC_DRIVER_ISR_PREEMPTION_ENABLED

Type: BooleanDefineOnly

Default value: 1 when all components connected to the VICs support preemption. 0 when any of the connected components don't support preemption.

Destination file: **system.h**

Description: Enables global interrupt preemption (nesting). When enabled (set to 1), the macro `ALTERA_VIC_DRIVER_ISR_PREEMPTION_ENABLED` is defined in **system.h**.

Two types of ISR preemption are available. This setting must be enabled along with other settings to enable specific types of preemption.

All preemption settings are dependant on whether the device drivers in your BSP support interrupt preemption. For more information about preemption, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Occurs: Once per VIC

altera_vic_driver.enable_preemption_into_new_register_set

Identifier: ALTERA_VIC_DRIVER_PREEMPTION_INTO_NEW_REGISTER_SET_ENABLED

Type: BooleanDefineOnly

Default value: 0

Destination file: **system.h**

Description: Enables interrupt preemption (nesting) if a higher priority interrupt is asserted while a lower priority ISR is executing, and that higher priority interrupt uses a different register set than the interrupt currently being serviced.

When this setting is enabled (set to 1), the macro `ALTERA_VIC_DRIVER_ISR_PREEMPTION_INTO_NEW_REGISTER_SET_ENABLED` is defined in **system.h** and the Nios II `config.ANI` (automatic nested interrupts) bit is asserted during system software initialization.

Use this setting to limit interrupt preemption to higher priority (RIL) interrupts that use a different register set than a lower priority interrupt that might be executing. This setting allows you to support some preemption while maintaining the lowest possible interrupt response time. However, this setting does not allow an interrupt at a higher priority (RIL) to preempt a lower priority interrupt if the higher priority interrupt is assigned to the same register set as the lower priority interrupt.

Occurs: Once per VIC

altera_vic_driver.enable_preemption_rs_*<n>*

Identifier: ALTERA_VIC_DRIVER_ENABLE_PREEMPTION_RS_*<n>*

Type: Boolean

Default value: 0

Destination file: **system.h**

Description: Enables interrupt preemption (nesting) if a higher priority interrupt is asserted while a lower priority ISR is executing, for all interrupts that target the specified register set number.

When this setting is enabled (set to 1), the vector table for each VIC utilizes a special interrupt funnel that manages preemption. All interrupts on all VIC instances assigned to that register set then use this funnel.

When a higher priority interrupt preempts a lower priority interrupt running in the same register set, the interrupt funnel detects this condition and saves the processor registers to the stack before calling the higher priority ISR. The funnel code restores registers and allows the lower priority ISR to continue running once the higher priority ISR completes.

Because this funnel contains additional overhead, enabling this setting increases interrupt response time substantially for all interrupts that target a register set where this type of preemption is enabled.

Use this setting if you must guarantee that a higher priority interrupt preempts a lower priority interrupt, and you assigned multiple interrupts at different priorities to the same Nios II shadow register set.

Occurs: Per register set; *<n>* refers to the register set number.

altera_vic_driver.linker_section

Identifier: ALTERA_VIC_DRIVER_LINKER_SECTION

Type: UnquotedString

Default value: .text

Destination file: **system.h**

Description: Specifies the linker section that each VIC's generated vector table and each interrupt funnel link to. The memory device that the specified linker section is mapped to must be connected to both the Nios II instruction and data masters in your SOPC Builder system.

Use this setting to link performance-critical code into faster memory. For example, if your system's code is in DRAM and you have an on-chip or tightly-coupled memory interface for interrupt handling code, assigning the VIC driver linker section to a section in that memory improves interrupt response time.

For more information about linker sections and the Nios II BSP Editor, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

Occurs: Once per VIC

altera_vic_driver.*<name>*.vec_size

Identifier: *<name>*_VEC_SIZE

Type: DecimalNumber

Default value: 16

Destination file: **system.h**

Description: Specifies the number of bytes in each vector table entry. Legal values are 16, 32, 64, 128, 256, and 512.

The generated VIC vector tables in the BSP require a minimum of 16 bytes per entry.

If you intend to write your own vector table or locate your ISR at the vector address, you can use a larger size.

The vector table's total size is equal to the number of interrupt ports on the VIC instance multiplied by the vector table entry size specified in this setting.

Occurs: Per instance; *<name>* refers to the component name you assign in SOPC Builder.

altera_vic_driver.*<name>*.irq*<n>*_rrs

Identifier: ALTERA_VIC_DRIVER_*<name>*_IRQ*<n>*_RRS

Type: DecimalNumber

Default value: Refer to "Default Settings for RRS and RIL".

Destination file: **system.h**

Description: Specifies the RRS for the interrupt connected to the corresponding port. Legal values are 1 to the number of shadow register sets defined for the processor.

Occurs: Per IRQ per instance; *<name>* refers to the VIC's name and *<n>* refers to the IRQ number that you assign in SOPC Builder. Refer to SOPC Builder to determine which IRQ numbers correspond to which components in your design.

altera_vic_driver.*<name>*.irq*<n>*_ril

Identifier: ALTERA_VIC_DRIVER_*<name>*_IRQ*<n>*_RIL

Type: DecimalNumber

Default value: Refer to "Default Settings for RRS and RIL".

Destination file: **system.h**

Description: Specifies the RIL for the interrupt connected to the corresponding port. Legal values are 0 to $2^{\text{RIL width}}$ -1.

Occurs: Per IRQ per instance; *<name>* refers to the VIC's name and *<n>* refers to the IRQ number that you assign in SOPC Builder. Refer to SOPC Builder to determine which IRQ numbers correspond to which components in your design.

altera_vic_driver.*<name>*.irq*<n>*_rnmi

Identifier: ALTERA_VIC_DRIVER_*<name>*_IRQ*<n>*_RNMI

Type: Boolean

Default value: 0

Destination file: **system.h**

Description: Specifies whether the interrupt port is a maskable or non-maskable interrupt (NMI). Legal values are 0 and 1. When set to 0, the port is maskable. NMIs cannot be disabled in hardware and there are several restrictions imposed for the RIL and RRS settings associated with any interrupt with NNI enabled.

Occurs: Per IRQ per instance; *<name>* refers to the VIC's name and *<n>* refers to the IRQ number that you assign in SOPC Builder. Refer to SOPC Builder to determine which IRQ numbers correspond to which components in your design.

Default Settings for RRS and RIL

The default assignment of RRS and RIL values for each interrupt assumes interrupt port 0 on the VIC instance attached to your processor is the highest priority interrupt, with successively lower priorities as the interrupt port number increases. Interrupt ports on other VIC instances connected through the first VIC's daisy chain interface are assigned successively lower priorities.

To make effective use of the VIC interrupt setting defaults, assign your highest priority interrupts to low interrupt port numbers on the VIC closest to the processor. Assign lower priority interrupts and interrupts that do not need exclusive access to a shadow register set, to higher interrupt port numbers, or to another daisy-chained VIC.

The following steps describe the algorithm for default RIL assignment:

1. The formula $2^{\text{RIL width}}$ -1 is used to calculate the maximum RIL value.
2. interrupt port 0 on the VIC connected to the processor is assigned the highest possible RIL.
3. The RIL value is decremented and assigned to each subsequent interrupt port in succession until the RIL value is 1.
4. The RILs for all remaining interrupt ports on all remaining VICs in the chain are assigned 1.

The following steps describe the algorithm for default RRS assignment:

1. The highest register set number is assigned to the interrupt with the highest priority.
2. Each subsequent interrupt is assigned using the same method as the default RIL assignment.

For example, consider a system with two VICs, VIC0 and VIC1. Each VIC has an RIL width of 3, and each has 4 interrupt ports. VIC0 is connected to the processor and VIC1 to the daisy chain interface on VIC0. The processor has 3 shadow register sets. Table 30–11 shows the default RRS and RIL assignments for this example.

Table 30–11. Default RRS and RIL Assignment Example

VIC	IRQ	RRS	RIL
0	0	3	7
0	1	2	6
0	2	1	5
0	3	1	4
1	0	1	3
1	1	1	2
1	2	1	1
1	3	1	1

VIC BSP Design Rules for Altera Hal Implementation

The VIC BSP settings allow for a large number of combinations. This list describes some basic design rules to follow to ensure a functional BSP:

- Each component's interrupt interface in your system should only be connected to one VIC instance per processor.
- The number of shadow register sets for the processor must be greater than zero.
- RRS values must always be greater than zero and less than or equal to the number of shadow register sets.
- RIL values must always be greater than zero and less than or equal to the maximum RIL.
- All RILs assigned to a register set must be sequential to avoid a higher priority interrupt overwriting contents of a register set being used by a lower priority interrupt.

  ☞ The Nios II BSP Editor uses the term "overlap condition" to refer to nonsequential RIL assignments.

- NMIs cannot share register sets with maskable interrupts.
- NMIs must have RILs set to a number equal to or greater than the highest RIL of any maskable interrupt. When equal, the NMIs must have a lower logical interrupt port number than any maskable interrupt.
- The vector table and funnel code section's memory device must connect to a data master and an instruction master.
- NMIs must use funnels with preemption disabled.
- When global preemption is disabled, enabling preemption into a new register set or per-register-set preemption might produce unpredictable results. Be sure that all interrupt service routines (ISR) used by the register set support preemption.
- Enabling register set preemption for register sets with peripherals that don't support preemption might result in unpredictable behavior.

RTOS Considerations

BSPs configured to use a real time operating system (RTOS) might have additional software linked into the HAL interrupt funnel code using the `ALT_OS_INT_ENTER` and `ALT_OS_INT_EXIT` macros. The exact nature and overhead of this code is RTOS-specific. Additional code adds to interrupt response and recovery time. Refer to your RTOS documentation to determine if such code is necessary.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*
- *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*
- *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

Table 30–12 shows the revision history for this chapter.

Table 30–12. Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	Initial release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

Section V. Test and Debug Peripherals

This section describes test and debug peripherals provided by Altera for SOPC Builder systems.

This section includes the following chapters:

- Chapter 31, Avalon-ST JTAG Interface Core
- Chapter 32, System ID Core
- Chapter 33, Performance Counter Core
- Chapter 34, Avalon Streaming Test Pattern Generator and Checker Cores
- Chapter 35, Avalon Streaming Data Pattern Generator and Checker Cores

☞ For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

31. Avalon-ST JTAG Interface Core

Core Overview

The Avalon® Streaming (Avalon-ST) JTAG Interface core enables communication between SOPC Builder systems and JTAG hosts via Avalon-ST interface. Data is serially transferred on the JTAG interface, and presented on the Avalon-ST interface as bytes.

The SPI Slave/JTAG to Avalon Master Bridge is an example of how this core is used. For more information about the bridge, refer to "SPI Slave/JTAG to Avalon Master Bridge Cores" on page 17–1

The Avalon-ST JTAG Interface core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

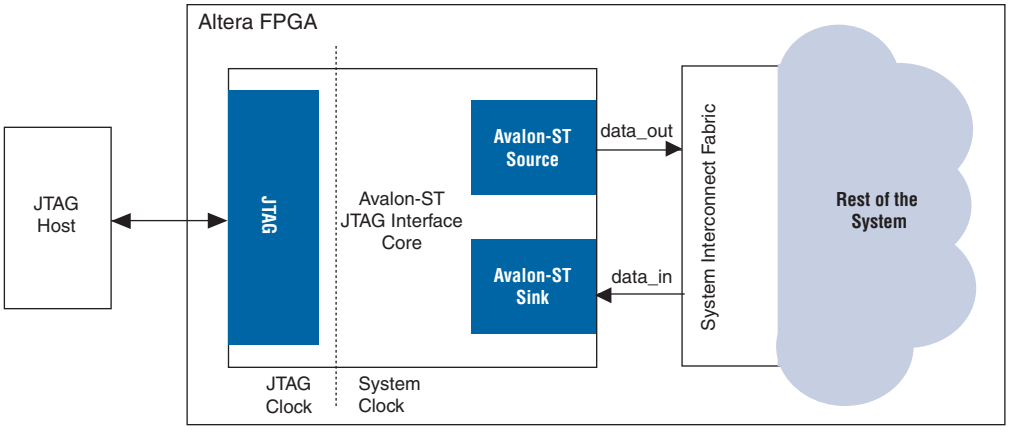
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 31–3
- "Device Support" on page 31–3

Functional Description

Figure 31–1 shows a block diagram of the Avalon-ST JTAG Interface core in a typical system configuration.

Figure 31–1. SOPC Builder System with an Avalon-ST JTAG Interface Core



Interfaces

Table 31–1 shows the properties of the Avalon-ST interfaces.

Table 31–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Only supported on the sink interface.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.
Error	Not used.
Packet	Not supported.

For more information about Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Special characters

Table 31–2 lists the special characters recognized by the core.

Table 31–2. Special Characters

Character	Description
`0x4a`	Idle. Idle characters are inserted into data streams when there is no data to send.
`0x4d`	Idle escape. An idle escape character is inserted into data stream when the data to send is a special character, followed by the data which is XORed with `0x20`.

Operation

The Avalon-ST JTAG Interface core accepts incoming data in bits on its JTAG interface and packs the bits into bytes. After each byte is formed, the core checks for the following special characters:

- `0x4a`—Idle character. The core drops the idle character.
- `0x4d`—Escape character. The core drops the escape character, and `XOR`s the following byte with `0x20`.

Each valid byte is then transferred to the core's Avalon-ST source interface. As there are no means to backpressure this interface, you must ensure that sufficient storage is in place to avoid data loss.

In the opposite direction, the core serializes each byte received on its Avalon-ST sink interface and sends the bits to the JTAG interface. If there is no data on the sink interface, the core sends out idle characters. If the data is a special character, the core inserts an escape character and `XOR`s the data with `0x20`.

The core supports four operation modes. From the system console, you can set the instruction register (IR) to enable the following supported modes:

- Normal mode—The core works as a bridge between a JTAG host and an SOPC Builder system. Set the IR to 0 to enable this mode.
- Loopback—Data received by the core is sent back to the host. Set the IR to 1 to enable this mode.

- Troubleshoot—The core retrieves the value of the system reset and clock signals, and return them to the JTAG host. Set the IR to 2 to enable this mode.

A TimeQuest SDC file (**.sdc**) is provided to cut any paths internal to the core.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST JTAG Interface core in SOPC Builder to add the core to a system. There are no user-configurable parameters for this core.

Device Support

The Avalon-ST JTAG Interface core supports all Altera® device families.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *"SPI Slave/JTAG to Avalon Master Bridge Cores" on page 17–1*

Document Revision History

Table 31–3 shows the revision history for this chapter.

Table 31–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

32. System ID Core

Core Overview

The system ID core with Avalon® interface is a simple read-only device that provides SOPC Builder systems with a unique identifier. Nios® II processor systems use the system ID core to verify that an executable program was compiled targeting the actual hardware image configured in the target FPGA. If the expected ID in the executable does not match the system ID core in the FPGA, it is possible that the software will not execute correctly.

This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 32–2
- "Instantiating the Core in SOPC Builder" on page 32–2
- "Software Programming Model" on page 32–2

Functional Description

The system ID core provides a read-only Avalon Memory-Mapped (Avalon-MM) slave interface. This interface has two 32-bit registers, as shown in Table 32–1. The value of each register is determined at system generation time, and always returns a constant value.

Table 32–1. System ID Core Register Map

Offset	Register Name	R/W	Description
0	`id`	R	A unique 32-bit value that is based on the contents of the SOPC Builder system. The id is similar to a check-sum value; SOPC Builder systems with different components, different configuration options, or both, produce different id values.
1	`timestamp`	R	A unique 32-bit value that is based on the system generation time. The value is equivalent to the number of seconds after Jan. 1, 1970.

There are two basic ways to use the system ID core:

- Verify the system ID before downloading new software to a system. This method is used by software development tools, such as the Nios II integrated development environment (IDE). There is little point in downloading a program to a target hardware system, if the program is compiled for different hardware. Therefore, the Nios II IDE checks that the system ID core in hardware matches the expected system ID of the software before downloading a program to run or debug.
- Check system ID after reset. If a program is running on hardware other than the expected SOPC Builder system, the program may fail to function altogether. If the program does not crash, it can behave erroneously in subtle ways that are difficult to debug. To protect against this case, a program can compare the expected system ID against the system ID core, and report an error if they do not match.

Device Support

The system ID core supports all Altera® device families.

Instantiating the Core in SOPC Builder

The System ID core has no user-configurable features. The `id` and `timestamp` register values are determined at system generation time based on the configuration of the SOPC Builder system and the current time. You can add only one system ID core to an SOPC Builder system, and its name is always `sysid`.

After system generation, you can examine the values stored in the `id` and `timestamp` registers by opening the MegaWizard™ interface for the System ID core. Hovering the mouse over the component in SOPC Builder also displays a tool-tip showing the values.

☞ Since a unique `timestamp` value is added to the System ID HDL file each time you generate the SOPC Builder system, the Quartus II software recompiles the entire system if you have added the system as a design partition.

Software Programming Model

This section describes the software programming model for the system ID core. For Nios II processor users, Altera provides the HAL system library header file that defines the System ID core registers.

The System ID core comes with the following software files. These files provide low-level access to the hardware. Application developers should not modify these files.

- **alt_avalon_sysid_regs.h**—Defines the interface to the hardware registers.
- **alt_avalon_sysid.c, alt_avalon_sysid.h**—Header and source files defining the hardware access functions.

Altera provides one access routine, `alt_avalon_sysid_test()`, that returns a value indicating whether the system ID expected by software matches the system ID core.

alt_avalon_sysid_test()

Prototype: `alt_32 alt_avalon_sysid_test(void)`

Thread-safe: No.

Available from ISR: Yes.

Include: **<altera_avalon_sysid.h>**

Description: Returns 0 if the values stored in the hardware registers match the values expected by software. Returns 1 if the hardware timestamp is greater than the software timestamp. Returns -1 if the software timestamp is greater than the hardware timestamp.

Document Revision History

Table 32–2 shows the revision history for this chapter.

Table 32–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	Added description to the Instantiating the Core in SOPC Builder section.	The SOPC Builder works with incremental compilation.
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

33. Performance Counter Core

Core Overview

The performance counter core with Avalon® interface enables relatively unobtrusive, real-time profiling of software programs. With the performance counter, you can accurately measure execution time taken by multiple sections of code. You need only add a single instruction at the beginning and end of each section to be measured.

The main benefit of using the performance counter core is the accuracy of the profiling results. Alternatives include the following approaches:

- GNU profiler, `gprof`—`gprof` provides broad low-precision timing information about the entire software system. It uses a substantial amount of RAM, and degrades the real-time performance. For many embedded applications, `gprof` distorts real-time behavior too much to be useful.
- Interval timer peripheral—The interval timer is less intrusive than `gprof`. It can provide good results for narrowly targeted sections of code.

The performance counter core is unobtrusive, requiring only a single instruction to start and stop profiling, and no RAM. It is appropriate for high-precision measurements of narrowly targeted sections of code.

For further discussion of all three profiling methods, refer to *AN 391: Profiling Nios II Systems*.

The performance counter core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. The core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera® device drivers enable the Nios II processor to use the performance counters.

This chapter contains the following sections:

- "Functional Description" on page 33–2
- "Device and Tools Support" on page 33–3
- "Instantiating the Core in SOPC Builder" on page 33–3
- "Hardware Simulation Considerations" on page 33–4
- "Software Programming Model" on page 33–4
- "Performance Counter API" on page 33–7

Functional Description

The performance counter core is a set of counters which track clock cycles, timing multiple sections of your software. You can start and stop these counters in your software, individually or as a group. You can read cycle counts from hardware registers.

The core contains two counters for every section:

- Time: A 64-bit clock cycle counter.
- Events: A 32-bit event counter.

Section Counters

Each 64-bit time counter records the aggregate number of clock cycles spent in a section of code. The 32-bit event counter records the number of times the section executes.

The performance counter core can have up to seven section counters.

Global Counter

The global counter controls all section counters. The section counters are enabled only when the global counter is running.

The 64-bit global clock cycle counter tracks the aggregate time for which the counters were enabled. The 32-bit global event counter tracks the number of global events, that is, the number of times the performance counter core has been enabled.

Register Map

The performance counter core has an Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to memory-mapped registers. Reading from the registers retrieves the current times and event counts. Writing to the registers starts, stops, and resets the counters. Table 33–1 shows the registers in detail.

Table 33–1. Performance Counter Core Register Map (Part 1 of 2)

Offset	Register Name	Bit Description		
		Read	Write	
		31 … 0	31 … 1	0
0	$T[0]_{lo}$	global clock cycle counter [31: 0]	*(1)*	0 = STOP 1 = RESET
1	$T[0]_{hi}$	global clock cycle counter [63:32]	*(1)*	0 = START
2	`Ev[0]`	global event counter	*(1)*	*(1)*
3	—	*(1)*	*(1)*	*(1)*
4	$T[1]_{lo}$	section 1 clock cycle counter [31:0]	*(1)*	0 = STOP
5	$T[1]_{hi}$	section 1 clock cycle counter [63:32]	*(1)*	0 = START
6	`Ev[1]`	section 1 event counter	*(1)*	*(1)*
7	—	*(1)*	*(1)*	*(1)*
8	$T[2]_{lo}$	section 2 clock cycle counter [31:0]	*(1)*	0 = STOP

Table 33–1. Performance Counter Core Register Map (Part 2 of 2)

Offset	Register Name	Bit Description		
		Read	Write	
		31 … 0	31 … 1	0
9	$T[2]_{hi}$	section 2 clock cycle counter [63:32]	*(1)*	0 = START
10	Ev[2]	section 2 event counter	*(1)*	*(1)*
11	—	*(1)*	*(1)*	*(1)*
. . .	. . .	. . .	. . .	. . .
4*n* + 0	$T[n]_{lo}$	section *n* clock cycle counter [31:0]	*(1)*	0 = STOP
4*n* + 1	$T[n]_{hi}$	section *n* clock cycle counter [63:32]	*(1)*	0 = START
4*n* + 2	Ev[n]	section *n* event counter	*(1)*	*(1)*
4*n* + 3	—	*(1)*	*(1)*	*(1)*

Note to Table 33–1:

(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

System Reset

After a system reset, the performance counter core is stopped and disabled, and all counters are set to zero.

Device and Tools Support

The performance counter core supports all Altera device families supported by SOPC Builder, and provides device drivers for the Nios II hardware abstraction layer (HAL) system library.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the performance counter core in SOPC Builder to specify the core's hardware features.

Define Counters

Choose the number of section counters you want to generate by selecting from the **Number of simultaneously-measured sections** list. The performance counter core may have up to seven sections. If you require more that seven sections, you can instantiate multiple performance counter cores.

Multiple Clock Domain Considerations

If your SOPC Builder system uses multiple clocks, place the performance counter core in the same clock domain as the CPU. Otherwise, it is not possible to convert cycle counts to seconds correctly.

Hardware Simulation Considerations

You can use this core in simulation with no special considerations.

Software Programming Model

The following sections describe the software programming model for the performance counter core.

Software Files

Altera provides the following software files for Nios II systems. These files define the low-level access to the hardware and provide control and reporting functions. Do not modify these files.

- **altera_avalon_performance_counter.h, altera_avalon_performance_counter.c**—The header and source code for the functions and macros needed to control the performance counter core and retrieve raw results.
- **perf_print_formatted_report.c**—The source code for simple profile reporting.

Using the Performance Counter

In a Nios II system, you can control the performance counter core with a set of highly efficient C macros, and extract the results with C functions.

API Summary

The Nios II application program interface (API) for the performance counter core consists of functions, macros and constants.

Functions and macros

Table 33–2 lists macros and functions for accessing the performance counter hardware structure.

Table 33–2. Performance Counter Macros and Functions

Name	Summary
`PERF_RESET()`	Stops and disables all counters, resetting them to 0.
`PERF_START_MEASURING()`	Starts the global counter and enables section counters.
`PERF_STOP_MEASURING()`	Stops the global counter and disables section counters.
`PERF_BEGIN()`	Starts timing a code section.
`PERF_END()`	Stops timing a code section.
`perf_print_formatted_report()`	Sends a formatted summary of the profiling results to `stdout`.
`perf_get_total_time()`	Returns the aggregate global profiling time in clock cycles.
`perf_get_section_time()`	Returns the aggregate time for one section in clock cycles.
`perf_get_num_starts()`	Returns the number of counter events.
`alt_get_cpu_freq()`	Returns the CPU frequency in Hz.

For a complete description of each macro and function, see "Performance Counter API" on page 33–7.

Hardware Constants

You can get the performance counter hardware parameters from constants defined in **system.h**. The constant names are based on the performance counter instance name, specified on the **System Contents** tab in SOPC Builder. Table 33–3 lists the hardware constants.

Table 33–3. Performance Counter Constants

Name *(1)*	Meaning
PERFORMANCE_COUNTER_BASE	Base address of core
PERFORMANCE_COUNTER_SPAN	Number of hardware registers
PERFORMANCE_COUNTER_HOW_MANY_SECTIONS	Number of section counters

Note to Table 33–3:

(1) Example based on instance name `performance_counter`.

Startup

Before using the performance counter core, invoke `PERF_RESET` to stop, disable and zero all counters.

Global Counter Usage

Use the global counter to enable and disable the entire performance counter core. For example, you might choose to leave profiling disabled until your software has completed its initialization.

Section Counter Usage

To measure a section in your code, surround it with the macros `PERF_BEGIN()` and `PERF_END()`. These macros consist of a single write to the performance counter core.

You can simultaneously measure as many code sections as you like, up to the number specified in SOPC Builder. See "Define Counters" on page 33–3 for details. You can start and stop counters individually, or as a group.

Typically, you assign one counter to each section of code you intend to profile. However, in some situations you may wish to group several sections of code in a single section counter. As an example, to measure general interrupt overhead, you can measure all interrupt service routines (ISRs) with one counter.

To avoid confusion, assign a mnemonic symbol for each section number.

Viewing Counter Values

Library routines allow you to retrieve and analyze the results. Use `perf_print_formatted_report()` to list the results to `stdout`, as shown in Example 33–1.

Example 33–1.

```
perf_print_formatted_report(
     (void *)PERFORMANCE_COUNTER_BASE, // Peripheral's HW base address
     alt_get_cpu_freq(),               // defined in "system.h"
     3,                                // How many sections to print
     "1st checksum_test",              // Display-names of sections
     "pc_overhead",
     "ts_overhead");
```

Example 33–2 creates a table similar to this result.

Example 33–2.

```
--Performance Counter Report--
Total Time: 2.07711 seconds (103855534 clock-cycles)
+-----------------+--------+-----------+---------------+-----------+
| Section         |    %   | Time (sec)| Time (clocks) |Occurrences|
+-----------------+--------+-----------+---------------+-----------+
|1st checksum_test|     50 |   1.03800 |      51899750 |         1 |
+-----------------+--------+-----------+---------------+-----------+
| pc_overhead     |1.73e-05|   0.00000 |            18 |         1 |
+-----------------+--------+-----------+---------------+-----------+
| ts_overhead     |4.24e-05|   0.00000 |            44 |         1 |
+-----------------+--------+-----------+---------------+-----------+
```

For full documentation of `perf_print_formatted_report()`, see "Performance Counter API" on page 33–7.

Interrupt Behavior

The performance counter core does not generate interrupts.

You can start and stop performance counters, and read raw performance results, in an interrupt service routine (ISR). Do not call the `perf_print_formatted_report()` function from an ISR.

☞ If an interrupt occurs during the measurement of a section of code, the time taken by the CPU to process the interrupt and return to the section is added to the measurement time. The same applies to context switches in a multithreaded environment. Your software must take appropriate measures to avoid or handle these situations.

Performance Counter API

This section describes the application programming interface (API) for the performance counter core.

For Nios II processor users, Altera provides routines to access the performance counter core hardware. These functions are specific to the performance counter core and directly manipulate low level hardware. The performance counter core cannot be accessed via the HAL API or the ANSI C standard library.

PERF_RESET()

Prototype: `PERF_RESET(p)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `p`—performance counter core base address.

Returns: —

Description: Macro `PERF_RESET()` stops and disables all counters, resetting them to 0.

PERF_START_MEASURING()

Prototype: `PERF_START_MEASURING(p)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `p`—performance counter core base address.

Returns: —

Description: Macro `PERF_START_MEASURING()` starts the global counter, enabling the performance counter core. The behavior of individual section counters is controlled by `PERF_BEGIN()` and `PERF_END()`. `PERF_START_MEASURING()` defines the start of a global event, and increments the global event counter. This macro is a single write to the performance counter core.

PERF_STOP_MEASURING()

Prototype: `PERF_STOP_MEASURING(p)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `p`—performance counter core base address.

Returns: —

Description: Macro `PERF_STOP_MEASURING()` stops the global counter, disabling the performance counter core. This macro is a single write to the performance counter core.

PERF_BEGIN()

Prototype: `PERF_BEGIN(p,n)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `p`—performance counter core base address.

`n`—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.

Returns: —

Description: Macro `PERF_BEGIN()` starts the timer for a code section, defining the beginning of a section event, and incrementing the section event counter. If you subsequently use `PERF_STOP_MEASURING()` and `PERF_START_MEASURING()` to disable and re-enable the core, the section counter will resume. This macro is a single write to the performance counter core.

PERF_END()

Prototype: `PERF_END(p,n)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `p`—performance counter core base address.

`n`—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.

Returns: —

Description: Macro `PERF_END()` stops timing a code section. The section counter does not run, regardless whether the core is enabled or not. This macro is a single write to the performance counter core.

perf_print_formatted_report()

Prototype:

```
int perf_print_formatted_report (
    void* perf_base,
    alt_u32 clock_freq_hertz,
    int num_sections,
    char* section_name_1, ...
    char* section_name_n)
```

Thread-safe: No.

Available from ISR: No.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `perf_base`—Performance counter core base address.

`clock_freq_hertz`—Clock frequency.

`num_sections`—The number of section counters to display. This must not exceed *<instance_name>*_HOW_MANY_SECTIONS.

`section_name_1 ... section_name_n`—The section names to display. The number of section names varies depending on the number of sections to display.

Returns: 0

Description: Function `perf_print_formatted_report()` reads the profiling results from the performance counter core, and prints a formatted summary table.

This function disables all counters. However, for predictable results in a multi-threaded or interrupt environment, invoke `PERF_STOP_MEASURING()` when you reach the end of the code to be measured, rather than relying on `perf_print_formatted_report()`.

perf_get_total_time()

Prototype: `alt_u64 perf_get_total_time(void* hw_base_address)`

Thread-safe: No.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `hw_base_address`—base address of performance counter core.

Returns: Aggregate global time in clock cycles.

Description: Function `perf_get_total_time()` reads the raw global time. This is the aggregate time, in clock cycles, that the performance counter core has been enabled. This function has the side effect of stopping the counters.

perf_get_section_time()

Prototype: `alt_u64 perf_get_section_time`
`(void* hw_base_address, int which_section)`

Thread-safe: No.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `hw_base_address`—performance counter core base address.

`which_section`—counter section number.

Returns: Aggregate section time in clock cycles.

Description: Function `perf_get_section_time()` reads the raw time for a given section. This is the time, in clock cycles, that the section has been running. This function has the side effect of stopping the counters.

perf_get_num_starts()

Prototype: `alt_u32 perf_get_num_starts`
`(void* hw_base_address, int which_section)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `hw_base_address`—performance counter core base address.

`which_section`—counter section number.

Returns: Number of counter events.

Description: Function `perf_get_num_starts()` retrieves the number of counter events (or times a counter has been started). If `which_section` = 0, it retrieves the number of global events (times the performance counter core has been enabled). This function does not stop the counters.

alt_get_cpu_freq()

Prototype: `alt_u32 alt_get_cpu_freq()`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters:

Returns: CPU frequency in Hz.

Description: Function `alt_get_cpu_freq()` returns the CPU frequency in Hz.

Referenced Documents

This chapter references the application note, *AN 391: Profiling Nios II Systems*.

Document Revision History

Table 33–4 shows the revision history for this chapter.

Table 33–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	Updated `perf_print_formatted_report()` to remove the restriction on using small C library.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Updated the parameter description of the function `perf_print_formatted_report()`.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

34. Avalon Streaming Test Pattern Generator and Checker Cores

Core Overview

The data generation and monitoring solution for Avalon® Streaming (Avalon-ST) consists of two components: a test pattern generator core that generates packetized or non-packetized data and sends it out on an Avalon-ST data interface, and a test pattern checker core that receives the same data and checks it for correctness.

The test pattern generator core can insert different error conditions, and the test pattern checker reports these error conditions to the control interface, each via an Avalon Memory-Mapped (Avalon-MM) slave.

Both cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

This chapter contains the following sections:

- "Resource Utilization and Performance"
- "Test Pattern Generator" on page 34–3
- "Test Pattern Checker" on page 34–5
- "Device Support" on page 34–6
- "Hardware Simulation Considerations" on page 34–6
- "Software Programming Model" on page 34–7
- "Test Pattern Generator API" on page 34–12
- "Test Pattern Checker API" on page 34–16

Resource Utilization and Performance

Resource utilization and performance for the test pattern generator and checker cores depend on the data width, number of channels, and whether the streaming data uses the optional packet protocol.

Table 34–1 provides estimated resource utilization and performance for the test pattern generator core.

Table 34–1. Test Pattern Generator Estimated Resource Utilization and Performance

No. of Channels	Datawidth (No. of 8-bit Symbols Per Beat)	Packet Support	Stratix® II and Stratix II GX			Cyclone® II			Stratix		
			f_{MAX} (MHz)	ALM Count	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)
1	4	Yes	284	233	560	206	642	560	202	642	560
1	4	No	293	222	496	207	572	496	245	561	496
32	4	Yes	276	270	912	210	683	912	197	707	912
32	4	No	323	227	848	234	585	848	220	630	848
1	16	Yes	298	361	560	228	867	560	245	896	560
1	16	No	340	330	496	230	810	496	228	845	496
32	16	Yes	295	410	912	209	954	912	224	956	912
32	16	No	269	409	848	219	842	848	204	912	848

Table 34–2 provides estimated resource utilization and performance for the test pattern checker core.

Table 34–2. Test Pattern Checker Estimated Resource Utilization and Performance

No. of Channels	Datawidth (No. of 8-bit Symbols Per Beat)	Packet Support	Stratix II and Stratix II GX			Cyclone II			Stratix		
			f_{MAX} (MHz)	ALM Count	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)
1	4	Yes	270	271	96	179	940	0	174	744	96
1	4	No	371	187	32	227	628	0	229	663	32
32	4	Yes	185	396	3616	111	875	3854	105	795	3616
32	4	No	221	363	3520	133	686	3520	133	660	3520
1	16	Yes	253	462	96	185	1433	0	166	1323	96
1	16	No	277	306	32	218	1044	0	192	1004	32
32	16	Yes	182	582	3616	111	1367	3584	110	1298	3616
32	16	No	218	473	3520	129	1143	3520	126	1074	3520

© July 2010 Altera Corporation

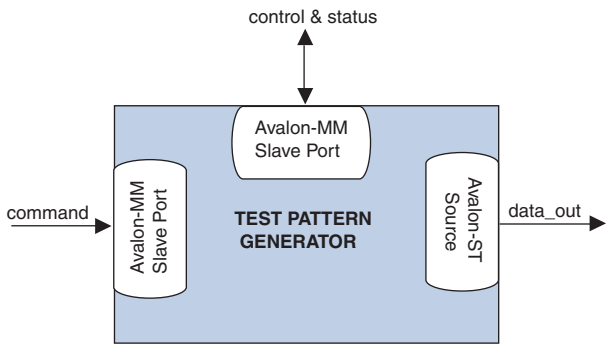
Test Pattern Generator

This section describes the hardware structure and functionality of the test pattern generator core.

Functional Description

The test pattern generator core accepts commands to generate data via an Avalon-MM command interface, and drives the generated data to an Avalon-ST data interface. You can parameterize most aspects of the Avalon-ST data interface such as the number of error bits and data signal width, thus allowing you to test components with different interfaces. Figure 34–1 shows a block diagram of the test pattern generator core.

Figure 34–1. Test Pattern Generator Core Block Diagram



The data pattern is determined by the following equation:
Symbol Value = Symbol Position in Packet `XOR` Data Error Mask. Non-packetized data is one long stream with no beginning or end.

The test pattern generator core has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register is used in conjunction with a pseudo-random number generator to throttle the data generation rate.

Command Interface

The command interface is a 32-bit Avalon-MM write slave that accepts data generation commands. It is connected to a 16-element deep FIFO, thus allowing a master peripheral to drive a number of commands into the test pattern generator core.

The command interface maps to the following registers: `cmd_lo` and `cmd_hi`. The command is pushed into the FIFO when the register `cmd_lo` (address 0) is written to. When the FIFO is full, the command interface asserts the `waitrequest` signal. You can create errors by writing to the register `cmd_hi` (address 1). The errors are only cleared when 0 is written to this register or its respective fields. See page "Test Pattern Generator Command Registers" on page 34–9 for more information on the register fields.

Control and Status Interface

The control and status interface is a 32-bit Avalon-MM slave that allows you to enable or disable the data generation as well as set the throttle.

This interface also provides useful generation-time information such as the number of channels and whether or not packets are supported.

Output Interface

The output interface is an Avalon-ST interface that optionally supports packets. You can configure the output interface to suit your requirements.

Depending on the incoming stream of commands, the output data may contain interleaved packet fragments for different channels. To keep track of the current symbol's position within each packet, the test pattern generator core maintains an internal state for each channel.

Instantiating the Test Pattern Generator in SOPC Builder

Use the MegaWizard™ interface for the test pattern generator core in SOPC Builder to configure the core. The following sections list the available options in the MegaWizard interface.

Functional Parameter

The functional parameter allows you to configure the test pattern generator as a whole: **Throttle Seed**—The starting value for the throttle control random number generator. Altera recommends a value which is unique to each instance of the test pattern generator and checker cores in a system.

Output Interface

You can configure the output interface of the test pattern generator core using the following parameters:

- **Number of Channels**—The number of channels that the test pattern generator core supports. Valid values are 1 to 256.
- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1 to 256. Example—For typical systems that carry 8-bit bytes, set this parameter to 8.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat. Valid values are 1 to 256.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Error Signal Width (bits)**—The width of the `error` signal on the output interface. Valid values are 0 to 31. A value of 0 indicates that the `error` signal is not used.

Test Pattern Checker

This section describes the hardware structure and functionality of the test pattern checker core.

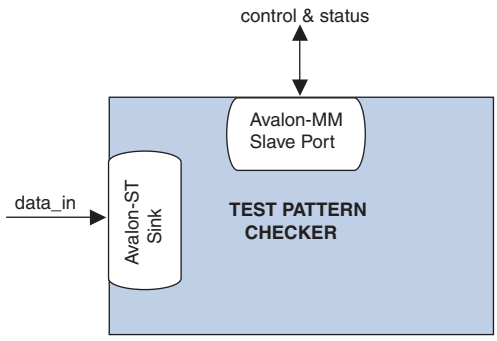
Functional Description

The test pattern checker core accepts data via an Avalon-ST interface, checks it for correctness against the same predetermined pattern used by the test pattern generator core to produce the data, and reports any exceptions to the control interface. You can parameterize most aspects of the test pattern checker's Avalon-ST interface such as the number of error bits and the data signal width, thus allowing you to test components with different interfaces.

The test pattern checker has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register controls the rate at which data is accepted.

Figure 34–2 shows a block diagram of the test pattern checker core.

Figure 34–2. Test Pattern Checker



The test pattern checker core detects exceptions and reports them to the control interface via a 32-element deep internal FIFO. Possible exceptions are data error, missing start-of-packet (SOP), missing end-of-packet (EOP) and signalled error.

As each exception occurs, an exception descriptor is pushed into the FIFO. If the same exception occurs more than once consecutively, only one exception descriptor is pushed into the FIFO. All exceptions are ignored when the FIFO is full. Exception descriptors are deleted from the FIFO after they are read by the control and status interface.

Input Interface

The input interface is an Avalon-ST interface that optionally supports packets. You can configure the input interface to suit your requirements.

Incoming data may contain interleaved packet fragments. To keep track of the current symbol’s position, the test pattern checker core maintains an internal state for each channel.

Control and Status Interface

The control and status interface is a 32-bit Avalon-MM slave that allows you to enable or disable data acceptance as well as set the throttle. This interface provides useful generation-time information such as the number of channels and whether the test pattern checker supports packets.

The control and status interface also provides information on the exceptions detected by the test pattern checker core. The interface obtains this information by reading from the exception FIFO.

Instantiating the Test Pattern Checker in SOPC Builder

Use the MegaWizard interface for the test pattern checker core in SOPC Builder to configure the core. The following sections list the available options in the MegaWizard interface.

Functional Parameter

The functional parameter allows you to configure the test pattern checker as a whole:
Throttle Seed—The starting value for the throttle control random number generator. Altera recommends a unique value to each instance of the test pattern generator and checker cores in a system.

Input Parameters

You can configure the input interface of the test pattern checker core using the following parameters:

- **Data Bits Per Symbol**—The number of bits per symbol for the input interface. Valid values are 1 to 256.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat. Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Number of Channels**—The number of channels that the test pattern checker core supports. Valid values are 1 to 256.
- **Error Signal Width (bits)**—The width of the `error` signal on the input interface. Valid values are 0 to 31. A value of 0 indicates that the `error` signal is not in use.

Device Support

The test pattern generator and checker cores support all Altera© device families.

Hardware Simulation Considerations

The test pattern generator and checker cores do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

This section describes the software programming model for the test pattern generator and checker cores.

HAL System Library Support

For Nios II processor users, Altera provides HAL system library drivers that enable you to initialize and access the test pattern generator and checker cores. Altera recommends you to use the provided drivers to access the cores instead of accessing the registers directly.

For Nios II IDE users, copy the provided drivers from the following installation folders to your software application directory:

- *<IP installation directory>* **/ip /sopc_builder_ip /altera_avalon_data_source/HAL**
- *<IP installation directory>***/ip/sopc_builder_ip/ altera_avalon_data_sink/HAL**

This instruction does not apply if you use the Nios II command-line tools.

Software Files

The following software files define the low-level access to the hardware, and provide the routines for the HAL device drivers. Application developers should not modify these files.

- Software files provided with the test pattern generator core:
  - **data_source_regs.h**—The header file that defines the test pattern generator's register maps.
  - **data_source_util.h, data_source_util.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.
- Software files provided with the test pattern checker core:
  - **data_sink_regs.h**—The header file that defines the core’s register maps.
  - **data_sink_util.h, data_sink_util.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.

Register Maps

This section describes the register maps for the test pattern generator and checker cores.

Test Pattern Generator Control and Status Registers

Table 34–3 shows the offset for the test pattern generator control and status registers. Each register is 32 bits wide.

Table 34–3. Test Pattern Generator Control and Status Register Map

Offset	Register Name
base + 0	`status`
base + 1	`control`
base + 2	`fill`

Table 34–4 describes the `status` register bits.

Table 34–4. Status Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	`ID`	RO	A constant value of `0x64`.
[23:16]	`NUMCHANNELS`	RO	The configured number of channels.
[30:24]	`NUMSYMBOLS`	RO	The configured number of symbols per beat.
[31]	`SUPPORTPACKETS`	RO	A value of 1 indicates packet support.

Table 34–5 describes the `control` register bits

Table 34–5. Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	`ENABLE`	RW	Setting this bit to 1 enables the test pattern generator core.
[7:1]	Reserved		
[16:8]	`THROTTLE`	RW	Specifies the throttle value which can be between 0–256, inclusively. This value is used in conjunction with a pseudorandom number generator to throttle the data generation rate. Setting `THROTTLE` to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	`SOFT RESET`	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

Table 34–6 describes the `fill` register bits.

Table 34–6. Fill Field Descriptions (Part 1 of 2)

Bit(s)	Name	Access	Description
[0]	`BUSY`	RO	A value of 1 indicates that data transmission is in progress, or that there is at least one command in the command queue.
[6:1]	Reserved		

Table 34–6. Fill Field Descriptions (Part 2 of 2)

Bit(s)	Name	Access	Description
[15:7]	FILL	RO	The number of commands currently in the command FIFO.
[31:16]	Reserved		

Test Pattern Generator Command Registers

Table 34–7 shows the offset for the command registers. Each register is 32 bits wide.

Table 34–7. Test Pattern Command Register Map

Offset	Register Name
base + 0	cmd_lo
base + 1	cmd_hi

Table 34–8 describes the cmd_lo register bits. The command is pushed into the FIFO only when the cmd_lo register is written to.

Table 34–8. cmd_lo Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	SIZE	RW	The segment size in symbols. Except for the last segment in a packet, the size of all segments must be a multiple of the configured number of symbols per beat. If this condition is not met, the test pattern generator core inserts additional symbols to the segment to ensure the condition is fulfilled.
[29:16]	CHANNEL	RW	The channel to send the segment on. If the channel signal is less than 14 bits wide, the low order bits of this register are used to drive the signal.
[30]	SOP	RW	Set this bit to 1 when sending the first segment in a packet. This bit is ignored when packets are not supported.
[31]	EOP	RW	Set this bit to 1 when sending the last segment in a packet. This bit is ignored when packets are not supported.

Table 34–9 describes the cmd_hi register bits.

Table 34–9. cmd_hi Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	SIGNALLED ERROR	RW	Specifies the value to drive the error signal. A non-zero value creates a signalled error.
[23:16]	DATA ERROR	RW	The output data is XORed with the contents of this register to create data errors. To stop creating data errors, set this register to 0.
[24]	SUPRESS SOP	RW	Set this bit to 1 to suppress the assertion of the startofpacket signal when the first segment in a packet is sent.
[25]	SUPRESS EOP	RW	Set this bit to 1 to suppress the assertion of the endofpacket signal when the last segment in a packet is sent.

Test Pattern Checker Control and Status Registers

Table 34–10 shows the offset for the control and status registers. Each register is 32 bits wide.

Table 34–10. Test Pattern Checker Control and Status Register Map

Offset	Register Name
base + 0	`status`
base + 1	`control`
base + 2	Reserved
base + 3	
base + 4	
base + 5	`exception_descriptor`
base + 6	`indirect_select`
base + 7	`indirect_count`

Table 34–11 describes the `status` register bits.

Table 34–11. Status Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	`ID`	RO	Contains a constant value of `0x65`.
[23:16]	`NUMCHANNELS`	RO	The configured number of channels.
[30:24]	`NUMSYMBOLS`	RO	The configured number of symbols per beat.
[31]	`SUPPORTPACKETS`	RO	A value of 1 indicates packet support.

Table 34–12 describes the `control` register bits.

Table 34–12. Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	`ENABLE`	RW	Setting this bit to 1 enables the test pattern checker.
[7:1]	Reserved		
[16:8]	`THROTTLE`	RW	Specifies the throttle value which can be between 0–256, inclusively. This value is used in conjunction with a pseudorandom number generator to throttle the data generation rate. Setting `THROTTLE` to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	`SOFT RESET`	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

Table 34–13 describes the `exception_descriptor` register bits. If there is no exception, reading this register returns 0.

Table 34–13. exception_descriptor Field Descriptions

Bit(s)	Name	Access	Description
[0]	`DATA ERROR`	RO	A value of 1 indicates that an error is detected in the incoming data.
[1]	`MISSINGSOP`	RO	A value of 1 indicates missing start-of-packet.
[2]	`MISSINGEOP`	RO	A value of 1 indicates missing end-of-packet.
[7:3]	Reserved		
[15:8]	`SIGNALLED ERROR`	RO	The value of the `error` signal.
[23:16]	Reserved		
[31:24]	`CHANNEL`	RO	The channel on which the exception was detected.

Table 34–14 describes the `indirect_select` register bits.

Table 34–14. indirect_select Field Descriptions

Bit	Bits Name	Access	Description
[7:0]	`INDIRECT CHANNEL`	RW	Specifies the channel number that applies to the `INDIRECT PACKET COUNT`, `INDIRECT SYMBOL COUNT`, and `INDIRECT ERROR COUNT` registers.
[15:8]	Reserved		
[31:16]	`INDIRECT ERROR`	RO	The number of data errors that occurred on the channel specified by `INDIRECT CHANNEL`.

Table 34–15 describes the `indirect_count` register bits.

Table 34–15. indirect_count Field Descriptions

Bit	Bits Name	Access	Description
[15:0]	`INDIRECT PACKET COUNT`	RO	The number of packets received on the channel specified by `INDIRECT CHANNEL`.
[31:16]	`INDIRECT SYMBOL COUNT`	RO	The number of symbols received on the channel specified by `INDIRECT CHANNEL`.

Test Pattern Generator API

This section describes the application programming interface (API) for the test pattern generator core. All API functions are currently not available from the interrupt service routine (ISR).

data_source_reset()

Prototype: `void data_source_reset(alt_u32 base);`

Thread-safe: No.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: `void.`

Description: This function resets the test pattern generator core including all internal counters and FIFOs. The control and status registers are not reset by this function.

data_source_init()

Prototype: `int data_source_init(alt_u32 base, alt_u32 command_base);`

Thread-safe: No.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

`command_base`—The base address of the command slave.

Returns: 1—Initialization is successful.

0—Initialization is unsuccessful.

Description: This function performs the following operations to initialize the test pattern generator core:

- Resets and disables the test pattern generator core.
- Sets the maximum throttle.
- Clears all inserted errors.

data_source_get_id()

Prototype: `int data_source_get_id(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The test pattern generator core's identifier.

Description: This function retrieves the test pattern generator core's identifier.

© July 2010 Altera Corporation

data_source_get_supports_packets()

Prototype: `int data_source_init(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: 1—Packets are supported.

0—Packets are not supported.

Description: This function checks if the test pattern generator core supports packets.

data_source_get_num_channels()

Prototype: `int data_source_get_num_channels(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The number of channels supported.

Description: This function retrieves the number of channels supported by the test pattern generator core.

data_source_get_symbols_per_cycle()

Prototype: `int data_source_get_symbols(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The number of symbols transferred in a beat.

Description: This function retrieves the number of symbols transferred by the test pattern generator core in each beat.

data_source_set_enable()

Prototype: `void data_source_set_enable(alt_u32 base, alt_u32 value);`

Thread-safe: No.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

`value`—The `ENABLE` bit is set to the value of this parameter.

Returns: `void.`

Description: This function enables or disables the test pattern generator core. When disabled, the test pattern generator core stops data transmission but continues to accept commands and stores them in the FIFO.

data_source_get_enable()

Prototype: `int data_source_get_enable(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The value of the `ENABLE` bit.

Description: This function retrieves the value of the `ENABLE` bit.

data_source_set_throttle()

Prototype: `void data_source_set_throttle(alt_u32 base, alt_u32 value);`

Thread-safe: No.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

`value`—The throttle value.

Returns: `void.`

Description: This function sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern generator sends data.

data_source_get_throttle()

Prototype: `int data_source_get_throttle(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The throttle value.

Description: This function retrieves the current throttle value.

data_source_is_busy()

Prototype: `int data_source_is_busy(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: 1—The test pattern generator core is busy.

0—The core is not busy.

Description: This function checks if the test pattern generator is busy. The test pattern generator core is busy when it is sending data or has data in the command FIFO to be sent.

© July 2010 Altera Corporation

data_source_fill_level()

Prototype: `int data_source_fill_level(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The number of commands in the command FIFO.

Description: This function retrieves the number of commands currently in the command FIFO.

data_source_send_data()

Prototype:

```
int data_source_send_data(alt_u32 cmd_base, alt_u16 channel,
alt_u16 size, alt_u32 flags, alt_u16 error, alt_u8
data_error_mask);
```

Thread-safe: No.

Include: **<data_source_util.h>**

Parameters: `cmd_base`—The base address of the command slave.

`channel`—The channel to send the data on.

`size`—The data size.

`flags`—Specifies whether to send or suppress SOP and EOP signals. Valid values are `DATA_SOURCE_SEND_SOP`, `DATA_SOURCE_SEND_EOP`, `DATA_SOURCE_SEND_SUPRESS_SOP` and `DATA_SOURCE_SEND_SUPRESS_EOP`.

`error`—The value asserted on the `error` signal on the output interface.

`data_error_mask`—This parameter and the data are `XOR`ed together to produce erroneous data.

Returns: Always returns 1.

Description: This function sends a data fragment to the specified channel.

If packets are supported, user applications must ensure the following conditions are met:

SOP and EOP are used consistently in each channel.

Except for the last segment in a packet, the length of each segment is a multiple of the data width.

If packets are not supported, user applications must ensure the following conditions are met:

No SOP and EOP indicators in the data.

The length of each segment in a packet is a multiple of the data width.

Test Pattern Checker API

This section describes the API for the test pattern checker core. The API functions are currently not available from the ISR.

data_sink_reset()

Prototype: `void data_sink_reset(alt_u32 base);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: `void.`

Description: This function resets the test pattern checker core including all internal counters.

data_sink_init()

Prototype: `int data_source_init(alt_u32 base);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: 1—Initialization is successful.

0—Initialization is unsuccessful.

Description: This function performs the following operations to initialize the test pattern checker core:

- Resets and disables the test pattern checker core.
- Sets the throttle to the maximum value.

data_sink_get_id()

Prototype: `int data_sink_get_id(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The test pattern checker core's identifier.

Description: This function retrieves the test pattern checker core's identifier.

data_sink_get_supports_packets()

Prototype: `int data_sink_init(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: 1—Packets are supported.

0—Packets are not supported.

Description: This function checks if the test pattern checker core supports packets.

data_sink_get_num_channels()

Prototype: `int data_sink_get_num_channels(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The number of channels supported.

Description: This function retrieves the number of channels supported by the test pattern checker core.

data_sink_get_symbols_per_cycle()

Prototype: `int data_sink_get_symbols(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The number of symbols received in a beat.

Description: This function retrieves the number of symbols received by the test pattern checker core in each beat.

data_sink_set enable()

Prototype: `void data_sink_set_enable(alt_u32 base, alt_u32 value);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

`value`—The `ENABLE` bit is set to the value of this parameter.

Returns: `void.`

Description: This function enables the test pattern checker core.

data_sink_get_enable()

Prototype: `int data_sink_get_enable(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The value of the `ENABLE` bit.

Description: This function retrieves the value of the `ENABLE` bit.

data_sink_set_throttle()

Prototype: `void data_sink_set_throttle(alt_u32 base, alt_u32 value);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

`value`—The throttle value.

Returns: `void.`

Description: This function sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern checker receives data.

data_sink_get_throttle()

Prototype: `int data_sink_get_throttle(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The throttle value.

Description: This function retrieves the throttle value.

data_sink_get_packet_count()

Prototype: `int data_sink_get_packet_count(alt_u32 base, alt_u32 channel);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

`channel`—Channel number.

Returns: The number of packets received on the given channel.

Description: This function retrieves the number of packets received on a given channel.

data_sink_get_symbol_count()

Prototype: `int data_sink_get_symbol_count(alt_u32 base, alt_u32 channel);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

`channel`—Channel number.

Returns: The number of symbols received on the given channel.

Description: This function retrieves the number of symbols received on a given channel.

data_sink_get_error_count()

Prototype: `int data_sink_get_error_count(alt_u32 base, alt_u32 channel);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

`channel`—Channel number.

Returns: The number of errors received on the given channel.

Description: This function retrieves the number of errors received on a given channel.

data_sink_get_exception()

Prototype: `int data_sink_get_exception(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The first exception descriptor in the exception FIFO.

0—No exception descriptor found in the exception FIFO.

Description: This function retrieves the first exception descriptor in the exception FIFO and pops it off the FIFO.

data_sink_exception_is_exception()

Prototype: `int data_sink_exception_is_exception(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor

Returns: 1—Indicates an exception.

0—No exception.

Description: This function checks if a given exception descriptor describes a valid exception.

data_sink_exception_has_data_error()

Prototype: `int data_sink_exception_has_data_error(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: 1—Data has errors.

0—No errors.

Description: This function checks if a given exception indicates erroneous data.

data_sink_exception_has_missing_sop()

Prototype: `int data_sink_exception_has_missing_sop(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: 1—Missing SOP.

0—Other exception types.

Description: This function checks if a given exception descriptor indicates missing SOP.

data_sink_exception_has_missing_eop()

Prototype: `int data_sink_exception_has_missing_eop(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: 1—Missing EOP.

0—Other exception types.

Description: This function checks if a given exception descriptor indicates missing EOP.

data_sink_exception_signalled_error()

Prototype: `int data_sink_exception_signalled_error(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: The signalled error value.

Description: This function retrieves the value of the signalled error from the exception.

data_sink_exception_channel()

Prototype: `int data_sink_exception_channel(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: The channel number on which the given exception occurred.

Description: This function retrieves the channel number on which a given exception occurred.

Document Revision History

Table 34–16 shows the revision history for this chapter.

Table 34–16. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Updated the section on HAL System Library Support.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

35. Avalon Streaming Data Pattern Generator and Checker Cores

Core Overview

The data generation and monitoring solution for Avalon® Streaming (Avalon-ST) interfaces consists of two components: a data pattern generator core that generates data patterns and sends it out on an Avalon-ST interface, and a data pattern checker core that receives the same data and checks it for correctness.

Both cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

This chapter contains the following sections:

- "Data Pattern Generator" on page 35–1
- "Data Pattern Checker" on page 35–4
- "Device Support" on page 35–6
- "Hardware Simulation Considerations" on page 35–6
- "Software Programming Model" on page 35–6

Data Pattern Generator

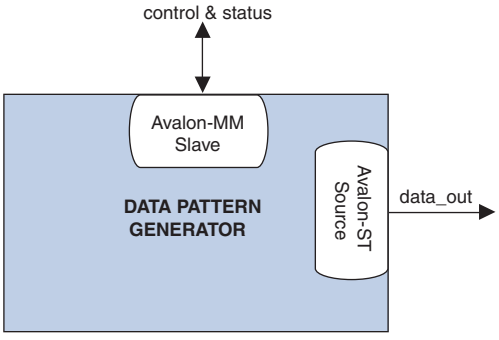
This section describes the hardware structure and functionality of the data pattern generator core.

Functional Description

The data pattern generator core accepts commands to generate and drive data onto a parallel Avalon-ST source interface.

Figure 35–1 shows a block diagram of the data pattern generator core.

Figure 35–1. Data Pattern Generator Core Block Diagram



You can configure the width of the output data signal to either 32-bit or 40-bit when instantiating the core.

You can configure this core to output 8-bit or 10-bit wide symbols. By default, the core generates 4 symbols per beat, which outputs 32-bit or 40-bit wide data to the Avalon-ST interfaces, respectively. The core's data format endianness is the most significant symbol first within a beat and the most significant bit first within a symbol. For example, when you configure the output data to 32-bit, bit 31 is the first data bit, followed by bit 30, and so forth. This interface's endianness may change in future versions of the core.

For smaller data widths, you can use the Avalon-ST Data Format Adapter for data width adaptation. The Avalon-ST Data Format Adapter converts the output from 4 symbols per beat, to 2 or 1 symbol per beat. In this way, the 32-bit output of the core can be adapted to a 16-bit or 8-bit output and the 40-bit output can be adapted to a 20-bit or 10-bit output.

For more information about the Avalon-ST Data Format Adapter, refer to the *Avalon Streaming Interconnect Components* chapter in volume 4 of the Quartus II Handbook.

Control and Status Interface

The control and status interface is an Avalon-MM slave that allows you to enable or disable the data generation. This interface also provides the run-time ability to choose data pattern and inject an error into the data stream.

Output Interface

The output interface is a parallel Avalon-ST interface. You can configure the data width at the output interface to suit your requirements.

Supported Data Patterns

The following data patterns are supported in the following manner, per beat. When the core is disabled or in idle state, the default pattern generated on the data output is 0×5555 (for 32-bit data width) or 0×55555 (for 40-bit data width).

Table 35–1 lists the supported data patterns for the data pattern generator core.

Table 35–1. Supported Data Patterns (Binary Encoding) *(Note 1)*

Pattern	32-bit	40-bit
PRBS-7	PRBS in parallel	PRBS in parallel
PRBS-15	PRBS in parallel	PRBS in parallel
PRBS-23	PRBS in parallel	PRBS in parallel
PRBS-31	PRBS in parallel	PRBS in parallel
High Frequency	10101010 x 4	1010101010 x 4
Low Frequency	11110000 x 4	1111100000 x 4

Note to Table 35–1:

(1) All PRBS patterns are seeded with 11111111.

This core does not support custom data patterns.

Inject Error

Errors can be injected into the data stream by controlling the `Inject Error` register bits in the register map (refer to Table 35–6 on page 35–7). When the inject error bit is set, one bit of error is produced by inverting the LSB of the next data beat.

If the inject error bit is set before the core starts generating the data pattern, the error bit is inserted in the first output cycle.

The `Inject Error` register bit is automatically reset after the error is introduced in the pipeline, so that the next error can be injected.

Preamble Mode

The preamble mode is used for synchronization or word alignment. When the preamble mode is set, the preamble control register sends the preamble character a specified number of times before the selected pattern is generated, so the word alignment block in the receiver can determine the word boundary in the bit stream.

The number of beats (`NumBeats`) determines the number of cycles to output the preamble character in the preamble mode. You can set the number of beats (`NumBeats`) in the preamble control register. The default setting is 0 and the maximum value is 255 beats. This mode can only be set when the data pattern generation core is disabled.

Instantiating the Data Pattern Generator in SOPC Builder

Use the MegaWizard™ interface for the data pattern generator core in SOPC Builder to configure the core. The following section lists the available option in the MegaWizard interface.

Output Parameter

You can configure the output interface of the data pattern generator core using the following parameter:

- **ST_DATA_W** — The width of the output data signal that the data pattern generator core supports. Valid values are 32 and 40.

Data Pattern Checker

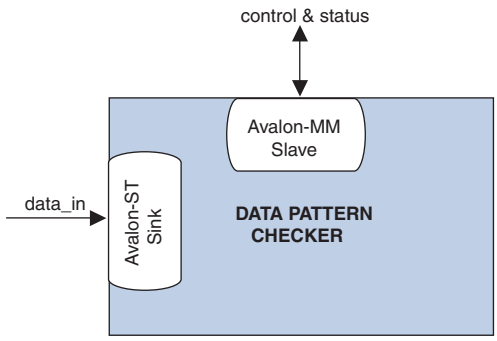
This section describes the hardware structure and functionality of the data pattern checker core.

Functional Description

The data pattern checker core accepts data via an Avalon-ST sink interface, checks it for correctness against the same predetermined pattern used by the data pattern generator core or other PRBS generators to produce the data, and reports any exceptions to the control interface.

Figure 35–2 shows a block diagram of the data pattern checker core.

Figure 35–2. Data Pattern Checker



You can configure the width of the output data signal to either 32-bit or 40-bit when instantiating the core. The chosen data width is not configurable during run time.

You can configure this core to output 8-bit or 10-bit wide symbols. By default, the core generates 4 symbols per beat, which outputs 32-bit or 40-bit wide data to the Avalon-ST interfaces, respectively. The core's data format endianness is the most significant symbol first within a beat and the most significant bit first within a symbol. For example, when you configure the output data to 32-bit, bit 31 is the first data bit, followed by bit 30, and so forth. This interface's endianness may change in future versions of the core.

If you configure the width of the output data to 32-bit, the core inputs four 8-bit wide symbols per beat. To achieve an 8-bit and 16-bit data width, you can use the Avalon-ST Data Format Adapter component to convert 4 symbols per beat to 1 or 2 symbols per beat.

Similarly, if you configure the width of the output data to 40-bit, the core inputs four 10-bit wide symbols per beat. The 10-bit and 20-bit input can be achieved by switching from 4 symbols per beat to 1 and 2 symbols per beat.

Control and Status Interface

The control and status interface is an Avalon-MM slave that allows you to enable or disable the pattern checking. This interface also provides the run-time ability to choose the data pattern and read the status signals.

Input Interface

The input interface is a parallel Avalon-ST interface. You can configure the data width at this interface to suit your requirements.

Supported Data Patterns

The following data patterns are supported in the following manner, per beat. When the core is disabled or in idle state, the default pattern generated on the data output is 0×5555 (for 32-bit data width) or 0×55555 (for 40-bit data width).

Table 35–2 lists the supported data patterns for the data pattern checker core.

Table 35–2. Supported Data Patterns (Binary Encoding)

Pattern	32-bit	40-bit
PRBS-7	PRBS in parallel	PRBS in parallel
PRBS-15	PRBS in parallel	PRBS in parallel
PRBS-23	PRBS in parallel	PRBS in parallel
PRBS-31	PRBS in parallel	PRBS in parallel
High Frequency	10101010 x 4	1010101010 x 4
Low Frequency	11110000 x 4	1111100000 x 4

Lock

The lock bit in the status register is asserted when 40 consecutive beats of correct data are received. The lock bit is deasserted and the receiver loses the lock when 40 consecutive beats of incorrect data are received.

Bit and Error Counters

The core has two 64-bit internal counters to keep track of the number of bits and number of error bits received. A snapshot has to be executed to update the `NumBits` and `NumErrors` registers with the current value from the internal counters.

A counter reset can be executed to reset both the registers and internal counters. If the counters are not being reset and the core is enabled, the internal counters continues the increment base on their current value.

☞ The internal counters only start to increment after a lock has been acquired.

Instantiating the Data Pattern Checker in SOPC Builder

Use the MegaWizard interface for the data pattern checker core in SOPC Builder to configure the core. The following section lists the available option in the MegaWizard interface.

Input Parameter

You can configure the input interface of the data pattern checker core using the following parameter:

- **ST_DATA_W** — The width of the input data signal that the data pattern checker core supports. Valid values are 32 and 40.

Device Support

The data pattern generator and checker cores support all Altera® device families.

Hardware Simulation Considerations

The data pattern generator and checker cores do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

This section describes the software programming model for the data pattern generator and checker cores.

Register Maps

This section describes the register maps for the data pattern generator and checker cores.

Data Pattern Generator Control Registers

Table 35–3 shows the offset for the control registers.

Table 35–3. Data Pattern Generator Register Map

Offset	Register Name
base + 0	`Enable`
base + 1	`Pattern Select`
base + 2	`Inject Error`
base + 3	`Preamble Control`
base + 4	`Preamble Character (Lower Bits)`
base + 5	`Preamble Character (Higher Bits)`

Table 35–4 describes the `Enable` register bits. This register enables or disables the pattern generation.

Table 35–4. Enable Field Descriptions *(Note 1)*

Bit(s)	Name	Access	Description
[0]	`EN`	RW	Setting this bit to 1 enables the data pattern generator core.
[31:1]	Reserved		

Note to Table 35–4:

(1) When the core is enabled, only the `Enable` register and the `Inject Error` register have write access. Write access to all other registers are ignored.The first valid data is observed from the Avalon-ST Source interface at the fourth cycle after the `Enable` bit is set. When the core is disabled, the final output is observed at the next clock cycle.

Table 35–5 describes the `pattern select` register bits.

Table 35–5. Pattern Select Field Descriptions *(Note 1)*

Bit(s)	Name	Access	Description
[0]	`PRBS7`	RW	Setting this bit to 1 outputs a PRBS 7 pattern with T [7, 6].
[1]	`PRBS15`	RW	Setting this bit to 1 outputs a PRBS 15 pattern with T [15, 14].
[2]	`PRBS23`	RW	Setting this bit to 1 outputs a PRBS 23 pattern with T [23, 18].
[3]	`PRBS31`	RW	Setting this bit to 1 outputs a PRBS 31 pattern with T [31, 28].
[4]	`HF`	RW	Setting this bit to 1 outputs a constant pattern of 0101010101… bits.
[5]	`LF`	RW	Setting this bit to 1 outputs a constant word pattern of 1111100000 for 10-bit words, or 11110000 for 8-bit words.
[31:8]	Reserved		

Note to Table 35–5:

(1) This register is one-hot encoded where only one of the pattern selector bits should be set to 1. For all other settings, the behaviors are undefined.

Table 35–6 describes the `Inject Error` register bits. This register allows you to set the error inject bit and insert one bit of error into the stream.

Table 35–6. Inject Error Field Descriptions *(Note 1)*

Bit(s)	Name	Access	Description
[0]	`IJ`	RW	Setting this bit to 1 injects error into the stream. If the `IJ` bit is set to 1 when the core is enabled, the bit resets itself to 0 at the next clock cycle when the error is injected.
[31:1]	Reserved		

Note to Table 35–6:

(1) The LSB of the data beat is flipped at the fourth clock cycle after the `IJ` bit is set (if not being backpressured by the sink when it is valid). The data beat that is injected with error might not be observed from the source if the core is disabled within the next two cycles after `IJ` bit is set to 1.

Table 35–7 describes the `Preamble Control` register bits. This register enables preamble and set the number of cycles to output the preamble character.

Table 35–7. Preamble Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	`EP`	RW	Setting this bit to 1, at the start of pattern generation, enables the preamble character to be sent for `NumBeats` cycles before switching over to the selected pattern.
[7:1]	Reserved		
[15:8]	`NumBeats`	RW	The number of beats to repeat the preamble character.
[31:16]	Reserved		

Table 35–8 describes the `Preamble Character (Lower Bits)` register bits. This register is for the user-defined preamble character (bit 0-31).

Table 35–8. Preamble Character Low Bits Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	`Preamble Character (Lower Bits)`	RW	Sets bit 31-0 for the preamble character to output.

Table 35–9 describes the `Preamble Character (Higher Bits)` register bits. This register is for the user-defined preamble character (bit 32-39) but is ignored if the `ST_DATA_W` value is set to 32.

Table 35–9. Preamble Character High Bits Field Descriptions

Bit(s)	Name	Access	Description
[7:0]	`Preamble Character (Higher Bits)`	RW	Sets bit 39-32 for the preamble character. This is ignored when the `ST_DATA_W` value is set to 32.
[31:8]	Reserved		

Data Pattern Checker Control and Status Registers

Table 35–10 shows the offset for the control and status registers.

Table 35–10. Data Pattern Checker Control and Status Register Map

Offset	Register Name
base + 0	`Status`
base + 1	`Pattern Set`
base + 2	`Counter Control`
base + 3	`NumBits (Lower Bits)`
base + 4	`NumBits (Higher Bits)`
base + 5	`NumErrors (Lower Bits)`
base + 6	`NumErrors (Higher Bits)`

Table 35–11 describes the `Status` register bits.

Table 35–11. Status Field Descriptions *(Note 1)*

Bit(s)	Name	Access	Description
[0]	`EN`	RW	Setting this bit to 1 enables pattern checking.
[1]	`LK`	R	Indicate lock status (writing to this bit has no effect).
[31:2]	Reserved		

Note to Table 35–11:

(1) When the core is enabled, only the `Status` register's `EN` bit and the `counter control` register have write access. Write access to all other registers are ignored.

Table 35–12 describes the `Pattern Select` register bits.

Table 35–12. Pattern Select Field Descriptions (Part 1 of 2) *(Note 1)*

Bit(s)	Name	Access	Description
[0]	`PRBS7`	RW	Setting this bit to 1 compares the data to a PRBS 7 pattern with T [7, 6].
[1]	`PRBS15`	RW	Setting this bit to 1 compares the data to a PRBS 15 pattern with T [15, 14].
[2]	`PRBS23`	RW	Setting this bit to 1 compares the data to a PRBS 23 pattern with T [23, 18].
[3]	`PRBS31`	RW	Setting this bit to 1 compares the data to a PRBS 31 pattern with T [31, 28].
[4]	`HF`	RW	Setting this bit to 1 compares the data to a constant pattern of 0101010101… bits.

Table 35–12. Pattern Select Field Descriptions (Part 2 of 2) *(Note 1)*

Bit(s)	Name	Access	Description
[5]	LF	RW	Setting this bit to 1 compares the data to a constant word pattern of 1111100000 for 10-bit words, or 11110000 for 8-bit words.
[31:8]	Reserved		

Note to Table 35–12:

(1) This register is one-hot encoded where only one of the pattern selector bits should be set to 1. For all other settings, the behaviors are undefined.

Table 35–13 describes the `Counter Control` register bits. This register snapshots and resets the NumBits, NumErrors, and also the internal counters.

Table 35–13. Counter Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	SN	W	Writing this bit to 1 captures the number of bits received and number of error bits received from the internal counters to the respective NumBits and NumErrors registers within the same clock cycle. Writing this bit to 1 after disabling the core will still capture the correct values from the internal counters to the NumBits and NumErrors registers.
[17]	RST	W	Writing this bit to 1 resets all internal counters and statistics. This bit resets itself automatically after the reset process. Re-enabling the core does not automatically reset the number of bits received and number of error bits received in the internal counter.
[31:18]	Reserved		

Table 35–14 describes the `NumBits (Lower Bits)` register bits. This register is the lower word of the 64-bit bit counter snapshot value. The register is reset when the component-reset is asserted or when the `RST` bit is set to 1.

Table 35–14. NumBits (Lower Word) Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	NumBits (Lower Bits)	R	Sets bit 31-0 for the `NumBits` (number of bits received).

Table 35–15 describes the `NumBits (Higher Bits)` register bits. This register is the higher word of the 64-bit bit counter snapshot value. The register is reset when the component-reset is asserted or when the `RST` bit is set to 1.

Table 35–15. NumBits (Higher Word) Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	NumBits (Higher Bits)	R	Sets bit 63-32 for the `NumBits` (number of bits received).

Table 35–16 describes the `NumErrors (Lower Bits)` register bits. This register is the lower word of the 64-bit error counter snapshot value. The register is reset when the component-reset is asserted or when the `RST` bit is set to 1.

Table 35–16. NumErrors (Lower Word) Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	`NumErrors (Lower Bits)`	R	Sets bit 31-0 for the `NumErrors` (number of error bits received).

Table 35–17 describes the `NumErrors (Higher Bits)` register bits. This register is the higher word of the 64-bit error counter snapshot value. The register is reset when the component-reset is asserted or when the `RST` bit is set to 1.

Table 35–17. NumErrors (Higher Word) Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	`NumErrors (Higher Bits)`	R	Sets bit 63-32 for the `NumErrors` (number of error bits received).

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Avalon Streaming Interconnect Components* chapter in volume 4 of the Quartus II Handbook.

Document Revision History

Table 35–18 shows the revision history for this chapter.

Table 35–18. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
January 2010 v9.1.1	Initial release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

Section VI. Clock Control Peripherals

This section describes clock control peripherals provided by Altera for SOPC Builder systems.

This section includes the following chapter:

- Chapter 36, PLL Cores

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

36. PLL Cores

Core Overview

The PLL cores, Avalon ALTPLL and PLL, provide a means of accessing the dedicated on-chip PLL circuitry in the Altera® Stratix®, except Stratix V, and Cyclone® series FPGAs. Both cores are a component wrapper around the Altera ALTPLL megafunction.

☞ The PLL core is scheduled for product obsolescence and discontinued support. Therefore, Altera recommends that you use the Avalon ALTPLL core in your designs.

The core takes an SOPC Builder system clock as its input and generates PLL output clocks locked to that reference clock.

The PLL cores support the following features:

- All PLL features provided by Altera's ALTPLL megafunction. The exact feature set depends on the device family.
- Access to status and control signals via Avalon Memory-Mapped (Avalon-MM) registers or top-level signals on the SOPC Builder system module.
- Dynamic phase reconfiguration in Stratix III and Stratix IV device families.

The PLL output clocks are made available in two ways:

- As sources to system-wide clocks in your SOPC Builder system.
- As output signals on your SOPC Builder system module.

For details about the ALTPLL megafunction, refer to the *ALTPLL Megafunction User Guide*.

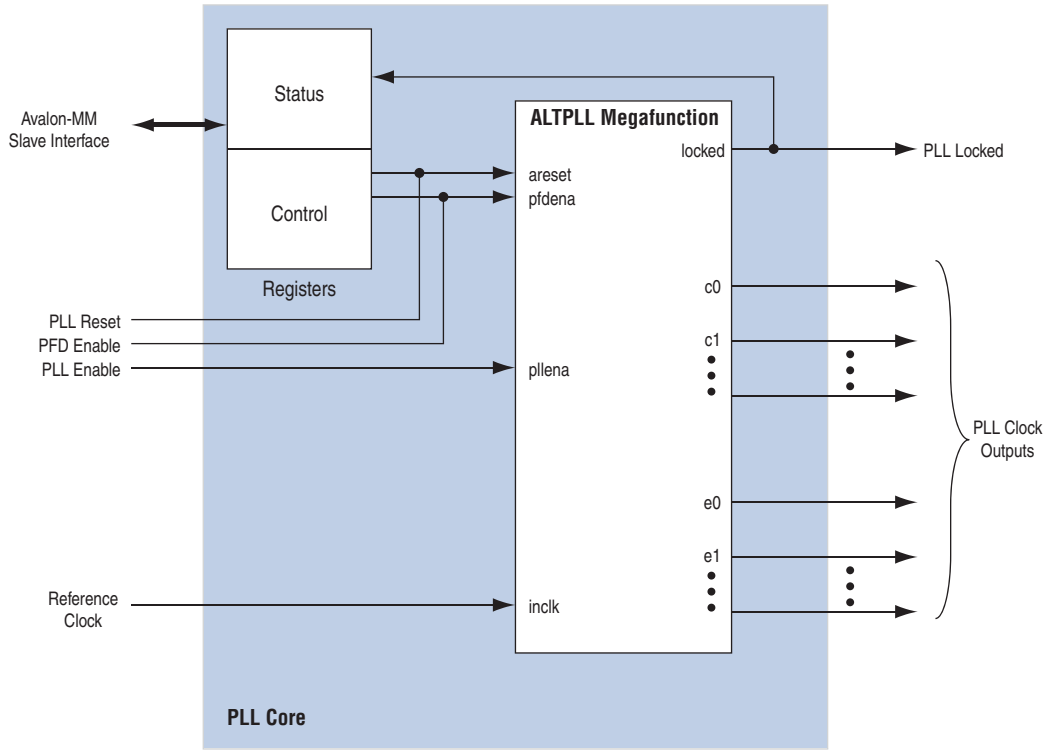
The PLL core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 36–3
- "Instantiating the Cores in SOPC Builder" on page 36–3
- "Hardware Simulation Considerations" on page 36–5
- "Register Definitions and Bit List" on page 36–5

Functional Description

Figure 36–1 shows a block diagram of the PLL cores and their connection to the PLL circuitry inside an Altera FPGA. The following sections describe the components of the core.

Figure 36–1. PLL Core Block Diagram



ALTPLL Megafunction

The PLL cores consist of an ALTPLL megafunction instantiation and an Avalon-MM slave interface. This interface can optionally provide access to status and control registers within the cores. The ALTPLL megafunction takes an SOPC Builder system clock as its reference, and generates one or more phase-locked loop output clocks.

Clock Outputs

Depending on the target device family, the ALTPLL megafunction can produce two types of output clock:

- internal (c)—clock outputs that can drive logic either inside or outside the SOPC Builder system module. Internal clock outputs can also be mapped to top-level FPGA pins. Internal clock outputs are available on all device families.
- external (e)—clock outputs that can only drive dedicated FPGA pins. They cannot be used as on-chip clock sources. External clock outputs are not available on all device families.

The Avalon ALTPLL core, however, does not differentiate the internal and external clock outputs and allows the external clock outputs to be used as on-chip clock sources.

To determine the exact number and type of output clocks available on your target device, refer to the *ALTPLL Megafunction User Guide*.

PLL Status and Control Signals

Depending on how the ALTPLL megafunction is parameterized, there can be a variable number of status and control signals. You can choose to export certain status and control signals to the top-level SOPC Builder system module. Alternatively, Avalon-MM registers can provide access to the signals. Any status or control signals which are not mapped to registers are exported to the top-level module. For details, refer to the "Instantiating the Cores in SOPC Builder" on page 36–3.

System Reset Considerations

At FPGA configuration, the PLL cores reset automatically. PLL-specific reset circuitry guarantees that the PLL locks before releasing reset for the overall SOPC Builder system module.

CAUTION Resetting the PLL resets the entire SOPC Builder system module.

Device Support

The PLL cores support all Altera device families.

Instantiating the Cores in SOPC Builder

The PLL cores contain an instantiation of the ALTPLL megafunction. The MegaWizard™ interface for the PLL cores allows you to configure the ALTPLL megafunction, and specify connections to selected status and control signals of the megafunction.

For details about using the ALTPLL MegaWizard Plug-In Manager, refer to the *ALTPLL Megafunction User Guide*.

Instantiating the Avalon ALTPLL Core

When you instantiate the Avalon ALTPLL core, the MegaWizard Plug-In Manager is automatically launched for you to parameterize the ALTPLL megafunction. There are no additional parameters that you can configure in SOPC Builder.

The `pfdena` signal of the ALTPLL megafunction is not exported to the top level of the SOPC Builder module. You can drive this port by writing to the `PFDENA` bit in the `control` register.

The `locked`, `pllena/extclkena`, and `areset` signals of the megafunction are always exported to the top level of the SOPC Builder module. You can read the `locked` signal and reset the core by manipulating respective bits in the registers. See "Register Definitions and Bit List" on page 36–5 for more information on the registers.

Instantiating the PLL Core

This section describes the options available in the MegaWizard interface for the PLL core in SOPC Builder.

PLL Settings Page

The **PLL Settings** page contains a button that launches the ALTPLL MegaWizard Plug-In Manager. Use the MegaWizard Plug-In Manager to parameterize the ALTPLL megafunction. The set of available parameters depends on the target device family.

You cannot click **Finish** in the PLL wizard nor configure the PLL interface until you parameterize the ALTPLL megafunction.

Interface Page

The **Interface** page configures the access modes for the optional advanced PLL status and control signals.

For each advanced signal present on the ALTPLL megafunction, you can select one of the following access modes:

- **Export**—Exports the signal to the top level of the SOPC builder system module.
- **Register**—Maps the signal to a bit in a status or control register.

The advanced signals are optional. If you choose not to create any of them in the ALTPLL MegaWizard Plug-In, the PLL's default behavior is as shown in Table 36–1.

You can specify the access mode for the advanced signals shown in Table 36–1. The ALTPLL core signals, not displayed in this table, are automatically exported to the top level of the SOPC Builder system module.

Table 36–1. ALTPLL Advanced Signal

ALTPLL Name	Input / Output	Avalon-MM PLL Wizard Name	Default Behavior	Description
`areset`	input	PLL Reset Input	The PLL is reset only at device configuration.	This signal resets the entire SOPC Builder system module, and restores the PLL to its initial settings.
`pllena`	input	PLL Enable Input	The PLL is enabled.	This signal enables the PLL. `pllena` is always exported.
`pfdena`	input	PFD Enable Input	The phase-frequency detector is enabled.	This signal enables the phase-frequency detector in the PLL, allowing it to lock on to changes in the clock reference.
`locked`	output	PLL Locked Output	—	This signal is asserted when the PLL is locked to the input clock.

CAUTION Asserting `areset` resets the entire SOPC Builder system module, not just the PLL.

Finish

Click **Finish** to insert the PLL into the SOPC Builder system. The PLL clock output(s) appear in the clock settings table on the SOPC Builder **System Contents** tab.

If the PLL has external output clocks, they appear in the clock settings table like other clocks; however, you cannot use them to drive components within the SOPC Builder system.

For details about using external output clocks, refer to the *ALTPLL Megafunction User Guide*.

The SOPC Builder automatically connects the PLL's reference clock input to the first available clock in the clock settings table.

If there is more than one SOPC Builder system clock available, verify that the PLL is connected to the appropriate reference clock.

Hardware Simulation Considerations

The HDL files generated by SOPC Builder for the PLL cores are suitable for both synthesis and simulation. The PLL cores support the standard SOPC Builder simulation flow, so there are no special considerations for hardware simulation.

Register Definitions and Bit List

Table 36–2 shows the register map for the PLL cores. Device drivers can control and communicate with the cores through two memory-mapped registers, `status` and `control`. The width of these registers are 32 bits in the Avalon ALTPLL core but only 16 bits in the PLL core.

In the PLL core, the `status` and `control` bits shown in Table 36–2 are present only if they have been created in the ALTPLL MegaWizard Plug-In Manager, and set to **Register** on the **Interface** page in the PLL wizard. These registers are always created in the Avalon ALTPLL core.

Table 36–2. PLL Cores Register Map

Offset	Register Name	R/W	Bit Description													
			31/15 *(2)*	30	29	…	9	8	7	6	5	4	3	2	1	0
0	`status`	R/O	*(1)*												`phasedone`	`locked`
1	`control`	R/W	*(1)*												`pfdena`	`areset`
2	`phase reconfig control`	R/W	`phase`		*(1)*			`counter_number`								
3	—	—	Undefined													

Notes to Table 36–2:

(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

(2) The registers are 32-bit wide in the Avalon ALTPLL core and 16-bit wide in the PLL core.

Status Register

Embedded software can access the PLL status via the `status` register. Writing to `status` has no effect. Table 36–3 describes the function of each bit.

Table 36–3. Status Register Bits

Bit Number	Bit Name	Value after reset	Description
0	`locked` *(2)*	1	Connects to the `locked` signal on the ALTPLL megafunction. The `locked` bit is high when valid clocks are present on the output of the PLL.
1	`phasedone` *(2)*	0	Connects to the `phasedone` signal on the ALTPLL megafunction. The `phasedone` output of the ALTPLL is synchronized to the system clock.
2:15/31 *(1)*	—	—	Reserved. Read values are undefined.

Note to Table 36–3:

(1) The `status` register is 32-bit wide in the Avalon ALTPLL core and 16-bit wide in the PLL core.

(2) Both the `locked` and `phasedone` outputs from the Avalon ALTPLL component are available as conduits and reflect the non-synchronized outputs from the ALTPLL.

Control Register

Embedded software can control the PLL via the `control` register. Software can also read back the status of control bits. Table 36–4 describes the function of each bit.

Table 36–4. Control Register Bits

Bit Number	Bit Name	Value after reset	Description
0	`areset`	0	Connects to the `areset` signal on the ALTPLL megafunction. Writing a 1 to this bit initiates a PLL reset.
1	`pfdena`	1	Connects to the `pfdena` signal on the ALTPLL megafunction. Writing a 0 to this bit disables the phase frequency detection.
2:15/31 *(1)*	—	—	Reserved. Read values are undefined. When writing, set reserved bits to zero.

Note to Table 36–4:

(1) The `control`register is 32-bit wide in the Avalon ALTPLL core and 16-bit wide in the PLL core.

Phase Reconfig Control Register

Embedded software can control the dynamic phase reconfiguration via the `phase reconfig control` register. Table 36–5 describes the function of each bit.

Table 36–5. Phase Reconfig Control Register Bits

Bit Number	Bit Name	Value after reset	Description
0:8	`counter_number`	—	A binary 9-bit representation of the counter that needs to be reconfigured. Refer to Table 36–6 for the counter selection.
9:29	—	—	Reserved. Read values are undefined. When writing, set reserved bits to zero.

Table 36–5. Phase Reconfig Control Register Bits

Bit Number	Bit Name	Value after reset	Description
30:31	`phase` *(1)*	—	01: Step up phase of `counter_number` 10: Step down phase of `counter_number` 00 and 11: No operation

Note to Table 36–5:

(1) Phase step up or down when set to 1 (only applicable to the Avalon ALTPLL core).

Table 36–6 lists the counter number and selection. For example, 100 000 000 selects counter C0 and 100 000 001 selects counter C1.

Table 36–6. Counter_Number Bits and Selection

Counter_Number [0:8]	Counter Selection
0 0000 0000	All output counters
0 0000 0001	M counter
> 0 0000 0001	Undefined
1 0000 0000	C0
1 0000 0001	C1
1 0000 0010	C2
...	...
1 0000 1000	C8
1 0000 1001	C9
> 1 0000 1001	Undefined

Referenced Documents

This chapter references the *ALTPLL Megafunction User Guide*.

Document Revision History

Table 36–7 shows the revision history for this chapter.

Table 36–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	No change from previous release.	—
November 2009 v9.1.0	Revised descriptions of register fields and bits.	Features added to the register map.
March 2009 v9.0.0	Added information on the new Avalon ALTPLL core.	A new PLL core, Avalon ALTPLL, is released and the chapter is updated accordingly to include the new core.

Table 36–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of this chapter, refer to the Quartus II Handbook Archive.

About this User Guide

This user guide provides comprehensive information about the Altera® Quartus® II design software, version 10.0.0.

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact *(Note 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
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Third-Party Software Product Information

Third-party software products described in this handbook are not Altera products, are licensed by Altera from third parties, and are subject to change without notice. Updates to these third-party software products may not be concurrent with Quartus II software releases. Altera has assumed responsibility for the selection of such third-party software products and its use in the Quartus II 9.0 software release. To the extent that the software products described in this handbook are derived from third-party software, no third party warrants the software, assumes any liability regarding use of the software, or undertakes to furnish you any support or information relating to the software. EXCEPT AS EXPRESSLY SET FORTH IN THE APPLICABLE ALTERA PROGRAM LICENSE SUBSCRIPTION AGREEMENT UNDER WHICH THIS SOFTWARE WAS PROVDED TO YOU, ALTERA AND THIRD-PARTY LICENSORS DISCLAIM ALL WARRANTIES WITH RESPECT TO THE USE OF SUCH THIRD-PARTY SOFTWARE CODE OR DOCUMENTATION IN THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT. For more information, including the latest available version of specific third-party software products, refer to the documentation for the software in question.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.