Quartus II Version 7.2 Handbook

Volume 5: Embedded Peripherals

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

QII5V5-7.2

Contents

Section II. Communication Peripherals

Chapter 8. JTAG UART Core

Chapter 9. UART Core

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook, Volume 5*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1.	SDRAM Controller Core	
	Revised:	*October 2007*
	Part number:	*NII51005-7.2.0*
Chapter 2.	CompactFlash Core	
	Revised:	*October 2007*
	Part number:	*QII55005-7.2.0*
Chapter 3.	Common Flash Interface Controller Core	
	Revised:	*October 2007*
	Part number:	*NII51013-7.2.0*
Chapter 4.	EPCS Device Controller Core	
	Revised:	*October 2007*
	Part number:	*NII51012-7.2.0*
Chapter 5.	On-Chip FIFO Memory Core	
	Revised:	*October 2007*
	Part number:	*QII55002-7.2.0*
Chapter 6.	Scatter-Gather DMA Controller Core	
	Revised:	*January 2008*
	Part number:	*QII55003-7.2.1*
Chapter 7.	DMA Controller Core	
	Revised:	*October 2007*
	Part number:	*NII51006-7.2.0*
Chapter 8.	JTAG UART Core	
	Revised:	*October 2007*
	Part number:	*NII51009-7.2.0*
Chapter 9.	UART Core	
	Revised:	*October 2007*
	Part number:	*NII51010-7.2.0*

Chapter 10.	SPI Core	
	Revised:	*October 2007*
	Part number:	*NII51011-7.2.0*
Chapter 11.	Optrex 16207 LCD Controller Core	
	Revised:	*October 2007*
	Part number:	*NII51019-7.2.0*
Chapter 12.	Video Sync Generator and Pixel Converter Cores	
	Revised:	*October 2007*
	Part number:	*QII55006-7.2.0*
Chapter 13.	Mutex Core	
	Revised:	*October 2007*
	Part number:	*NII51020-7.2.0*
Chapter 14.	Mailbox Core	
	Revised:	*October 2007*
	Part number:	*NII53001-7.2.0*
Chapter 15.	PIO Core	
	Revised:	*October 2007*
	Part number:	*NII51007-7.2.0*
Chapter 16.	Timer Core	
	Revised:	*October 2007*
	Part number:	*NII51008-7.2.0*
Chapter 17.	System ID Core	
	Revised:	*October 2007*
	Part number:	*NII51014-7.2.0*
Chapter 18.	PLL Core	
	Revised:	*October 2007*
	Part number:	*NII53002-7.2.0*
Chapter 19.	Performance Counter Core	
	Revised:	*October 2007*
	Part number:	*QII55001-7.2.0*
Chapter 20.	Avalon Streaming Channel Multiplexer and Demultiplexer Cores	
	Revised:	*October 2007*
	Part number:	*QII55004-7.2.0*
Chapter 21.	Avalon Streaming Test Pattern Generator and Checker Cores	
	Revised:	*October 2007*
	Part number:	*QII55007-7.2.0*

About This Handbook

Introduction

This volume describes intellectual property (IP) cores provided by Altera® for embedded systems design. These cores are installed with the Quartus® II software, and you can use them free of charge in Altera devices. Each core is SOPC Builder ready and can be instantiated in any SOPC Builder system. Most cores provide software driver support for the Altera Nios® II processor, and work seemlessly in Nios II systems.

Each chapter provides complete reference for a core, including the following information:

- Hardware structure
- Features and interface(s) to the core
- Available options when instantiating the core in SOPC Builder
- Hardware simulation considerations, if any
- Software programming model, including a description of the registers and driver functions.
- Device and tools support

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

This document uses the typographic conventions shown in the following table.

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, filenames, filename extensions, and software utility names are shown in bold type. Examples: **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 about a particular topic.

Section I. Memory Peripherals

This section describes memory components and interfaces provided by Altera®. These components provide access to on-chip or off-chip memory for SOPC Builder systems.

See *About This Handbook* for further details.

This section includes the following chapters:

☞ 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

NII51005-7.2.0

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® FPGA 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 FPGA, the core presents an Avalon-MM slave port that appears as linear memory (that is, 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.

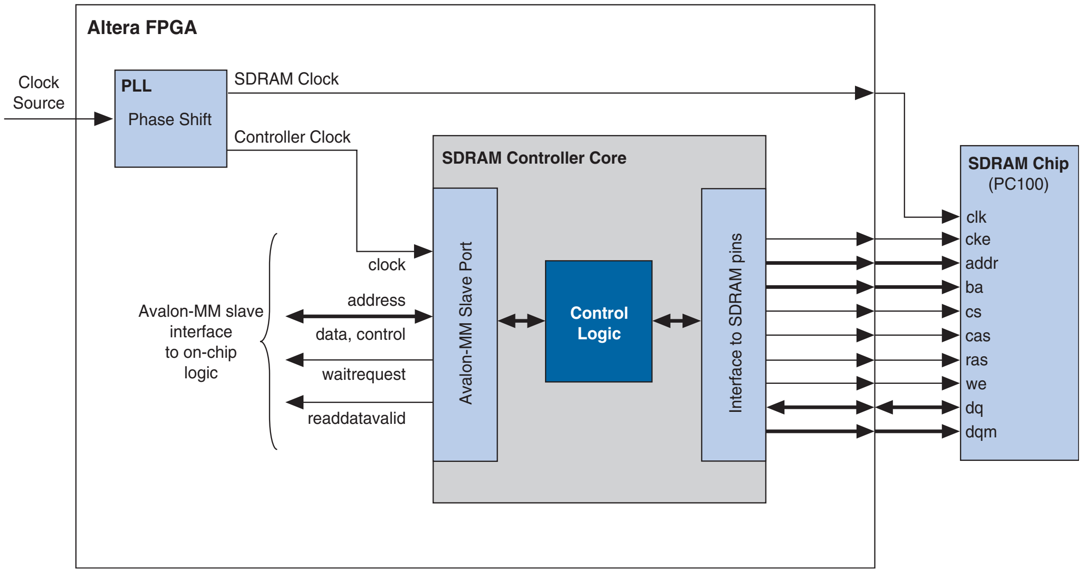
The SDRAM controller core with Avalon interface is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 1–2
- "Device and Tools Support" on page 1–6
- "Instantiating the Core in SOPC Builder" on page 1–6
- "Hardware Simulation Considerations" on page 1–9
- "Software Programming Model" on page 1–13
- "Clock, PLL and Timing Considerations" on page 1–13

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 Memory-Mapped Interface Specification.*

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 FPGA.

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 1–6 for details. The electrical characteristics of the FPGA pins depend on both the target device family and the assignments made in the Quartus® II software. Some FPGA families support a wider range of electrical standards, and therefore are capable of interfacing with a greater variety of SDRAM chips. For details, see the handbook for the target FPGA family.

Synchronizing Clock and Data Signals

The clock for the SDRAM chip (hereafter "SDRAM clock") must be driven at the same frequency as the clock for the Avalon-MM interface on the SDRAM controller (hereafter "controller clock"). As in all synchronous design, 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 1–13 for details.

For more information about instantiating a PLL in your SOPC Builder system, refer to the *PLL Core* chapter in volume 5 of the *Quartus II Handbook*. 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. It is not possible to drop A0 for memories when the smallest access size is 16 bits or A0-A1 when the smallest access size is 32 bits.

Board Layout and Pinout Considerations

When making decisions about the board layout and FPGA pinout, try to minimize the skew between the SDRAM signals. For example, when assigning the FPGA 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 below.

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 for row closings and openings.

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 requires row open and close overhead cycles. 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 FPGA

The target FPGA 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 FPGAs. However, the core might not achieve 100 MHz performance in all Altera FPGA 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 and Tools Support

The SDRAM Controller with Avalon interface core supports all Altera FPGA families. Different FPGA 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, see the handbook for the target FPGA family.

Instantiating the Core in SOPC Builder

Designers use the MegaWizard® Plug-In Manager interface for the SDRAM controller in SOPC Builder to specify hardware features and simulation features. The SDRAM controller MegaWizard interface 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 designers 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		checked (yes), unchecked (no)	No	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 pulldown menu.
Include a functional memory model in the system testbench		Yes, No	Yes	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 1–9.

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.

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

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, then 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, the designer is 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. 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

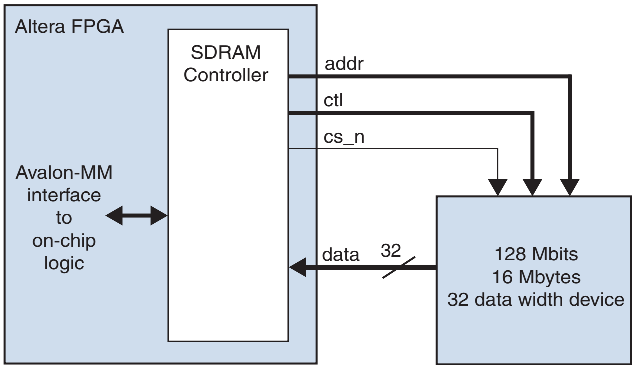


Figure 1–3 shows two 64-Mbit SDRAM chips, each with 16-bit data. Address and control signals connect in parallel to both chips. Note that chipselect (`cs_n`) is shared by the chips. Each chip provides half of the 32-bit data bus. The result is a logical 128-Mbit (16-Mbyte) 32-bit data memory.

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

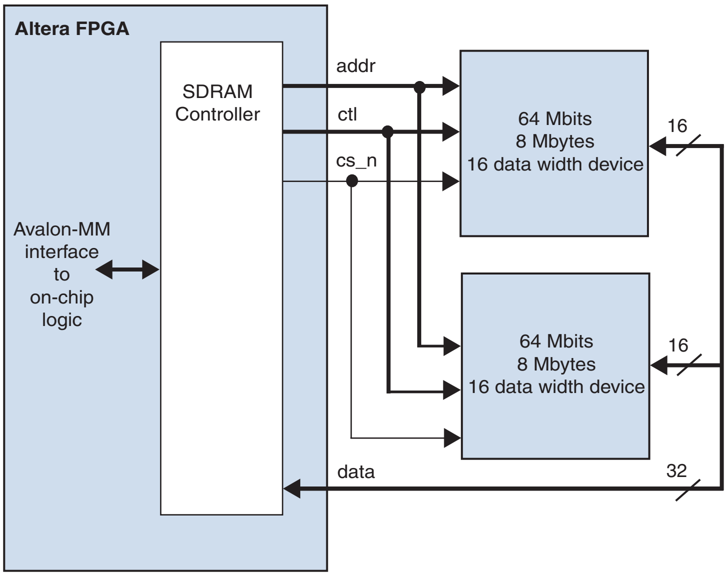
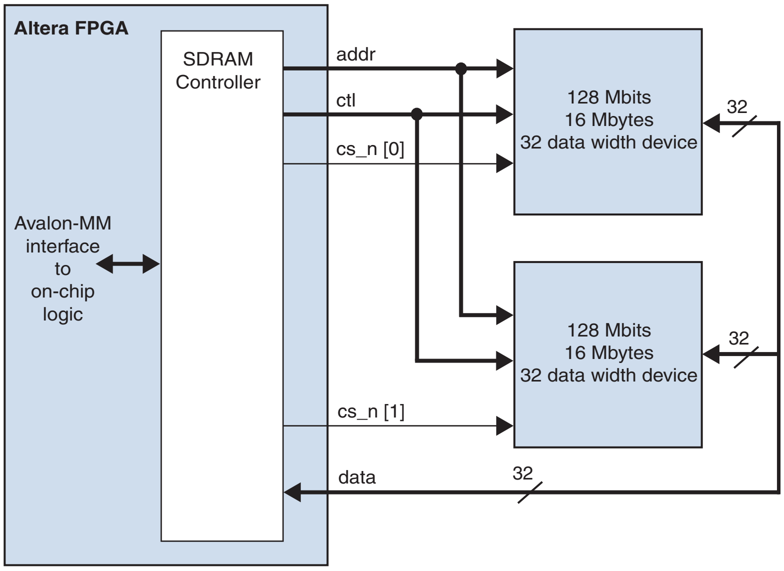

Altera FPGA
SDRAM Controller
Avalon-MM interface to on-chip logic
addr
ctl
cs_n
64 Mbits 8 Mbytes 16 data width device
16
64 Mbits 8 Mbytes 16 data width device
16
data
32

Figure 1–4 shows two 128-Mbit SDRAM chips, each with 32-bit data. 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


Altera FPGA
SDRAM Controller
Avalon-MM interface to on-chip logic
addr
ctl
cs_n [0]
cs_n [1]
data
32
128 Mbits 16 Mbytes 32 data width device
128 Mbits 16 Mbytes 32 data width device
32
32

Software Programming Model

The SDRAM controller behaves like simple memory when accessed via the Avalon-MM interface. There are no software-configurable settings, and there are 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 FPGAs, 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 FPGA and SDRAM I/O pins — I/O timing parameters vary based on device family and speed grade.
- Pin location on the FPGA — FPGA 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 FPGA 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. Furthermore, 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, then 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

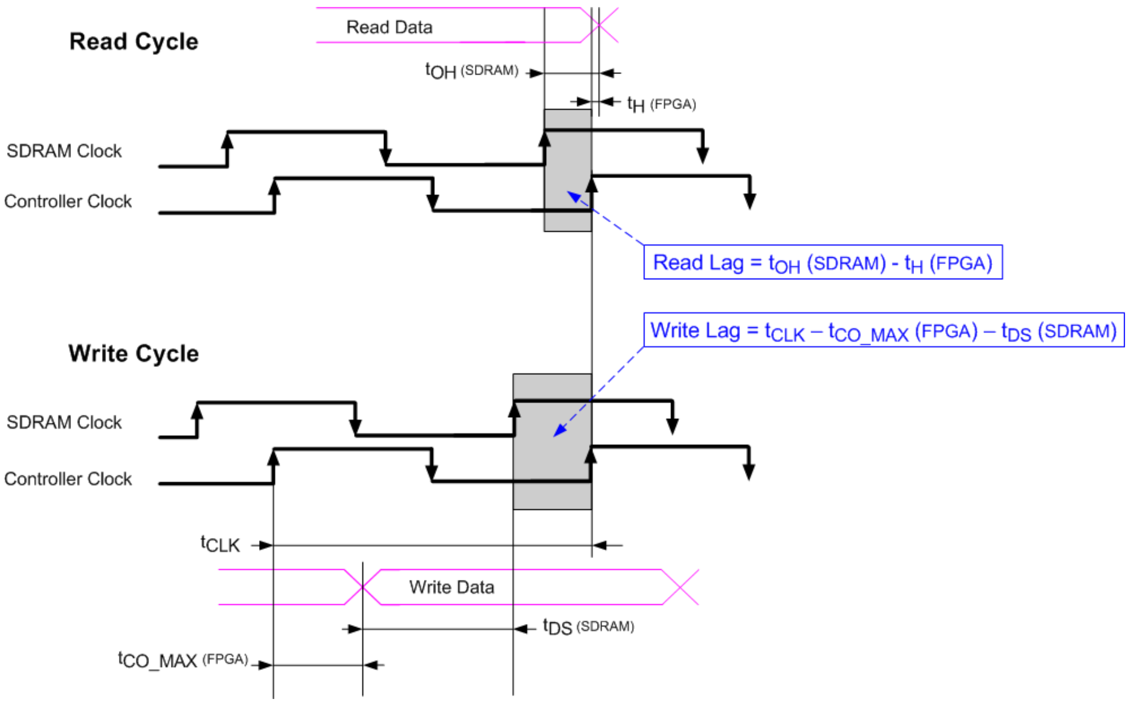
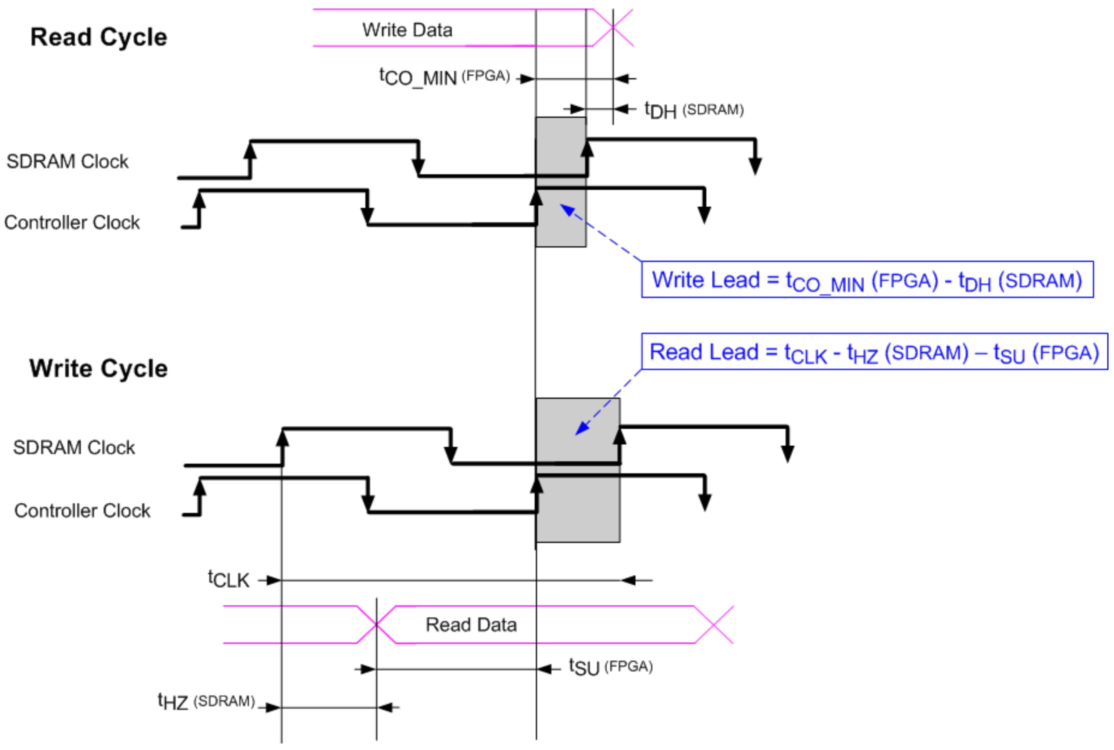


Figure 1–6. Calculating the Maximum SDRAM Clock Lead



Example Calculation

This section demonstrates a calculation of the signal window for a Micron MT48LC4M32B2-7 SDRAM chip and an FPGA design targeting an Altera Stratix II EP2S60F672C5 FPGA. This example uses a CAS latency (CL) of 3 cycles, and a clock frequency of 50 MHz. All SDRAM signals on the FPGA 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

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	
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 FPGA timing information, obtained from the Timing Analyzer section of the Quartus II Compilation Report. The values in Table 1–4 are the maximum or minimum values among all FPGA pins related to the SDRAM. The variance in timing between the SDRAM pins on the FPGA 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	`tCLK`	20
Minimum clock-to-output time	`tCO_MIN`	2.399
Maximum clock-to-output time	`tCO_MAX`	2.477
Maximum hold time after clock	`tH_MAX`	–5.607
Maximum setup time before clock	`tSU_MAX`	5.936

You must compile the design in the Quartus II software to obtain the I/O timing information for the FPGA 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.

With the values from Tables 1–3 and Table 1–4 you can perform the calculations from Figures 1–5 and 1–6, as shown below.

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

(1)

$$\text{Read Lag} = t_{OH}(\text{SDRAM}) - t_{H\_MAX}(\text{FPGA})$$

$$\text{Read Lag} = 2.5\,\text{ns} - (-5.607\,\text{ns})$$

$$\text{Read Lag} = 8.107\,\text{ns}$$

or

(2)

$$\text{Write Lag} = t_{CLK} - t_{CO\_MAX}(\text{FPGA}) - t_{DS}(\text{SDRAM})$$

$$\text{Write Lag} = 20\,\text{ns} - 2.477\,\text{ns}$$

$$\text{Write Lag} = 15.523\,\text{ns}$$

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

(3) $$\text{Read Lead} = t_{\text{CO\_MIN}}(\text{FPGA}) - t_{\text{DH}}(\text{SDRAM})$$

$$\text{Read Lead} = 2.399\,\text{ns} - 1.0\,\text{ns}$$

$$\text{Read Lead} = 1.399\,\text{ns}$$

or

(4) $$\text{Write Lead} = t_{\text{CLK}} - t_{\text{HZ}(3)}(\text{SDRAM}) - t_{\text{SU\_MAX}}(\text{FPGA})$$

$$\text{Write Lead} = 20\,\text{ns} - 5.5\,\text{ns} - 5.936\,\text{ns}$$

$$\text{Write Lead} = 8.564\,\text{ns}$$

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 $(-8.107 + 1.399) \div 2 = -3.35\,\text{ns}$.

Referenced Documents

This chapter references the following documents:

- *Avalon Memory-Mapped Interface Specification*
- *PLL Core* chapter in volume 5 the *Quartus II Handbook*
- *AN 351: Simulating Nios II Processor Designs*
- *altpll Megafunction User Guide*

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
October 2007 v7.2.0	No change from previous release.	—
May 2007 v7.1.0	● Updated description of Parameter Settings Memory Profile page to reflect new mechanism for for sharing pins via a tristate bridge. ● Added table of contents to Overview section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies. ● Changed old “Avalon switch fabric” term to “system interconnect fabric.” ● Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	Chapter title changed, but no change in content from previous release.	—
December 2005 v5.1.1	● Updated Figure 1-1. ● Updated sections “Off-Chip SDRAM Interface” and “Board Layout and Pinout Considerations.” ● Added section “Clock, PLL and Timing Considerations.”	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

2. CompactFlash Core

QII55005-7.2.0

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 CompactFlash core also provides a register-mapped 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.

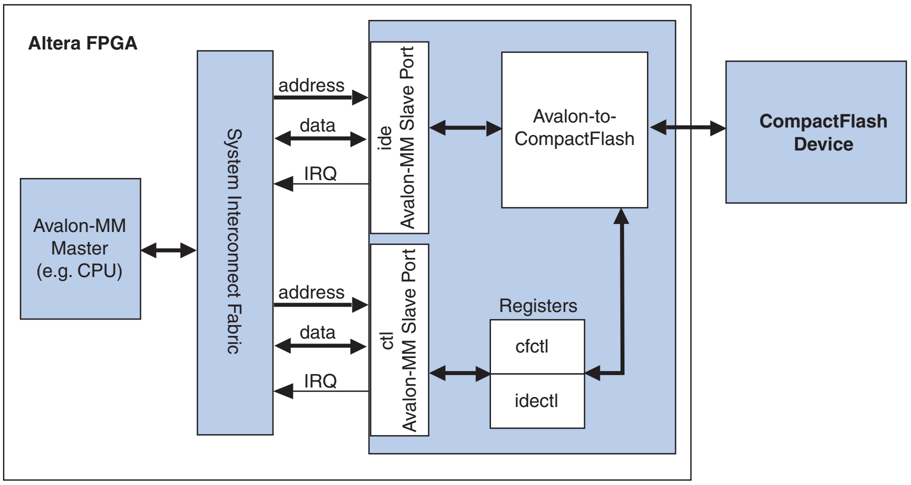
The CompactFlash core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

This chapter contains the following sections:

Functional Description

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 at www.compactflash.org.

Instantiating the Core in SOPC Builder

Use the MegaWizard® Plug-In Manager 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
`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

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

CompactFlash Interface Signal Name	Pin Type	CompactFlash Pin Number
`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 and Tools Support

The CompactFlash interface core supports all Altera FPGA 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 2–5.

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 control 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.

Referenced Documents

This chapter references the *Avalon Memory-Mapped Interface Specification*.

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
October 2007 v7.2.0	Initial release.	—

3. Common Flash Interface Controller Core

NII51013-7.2.0

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) and/or the ANSI C standard library functions for file I/O. For details about how to read and write flash using the HAL API, refer to the *Nios II Software Developer's Handbook*.

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® FPGA. For details, refer to the *Nios II Flash Programmer User Guide*.

Further information about the Common Flash Interface specification is available at **www.intel.com/design/flash/swb/cfi.htm**. 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.

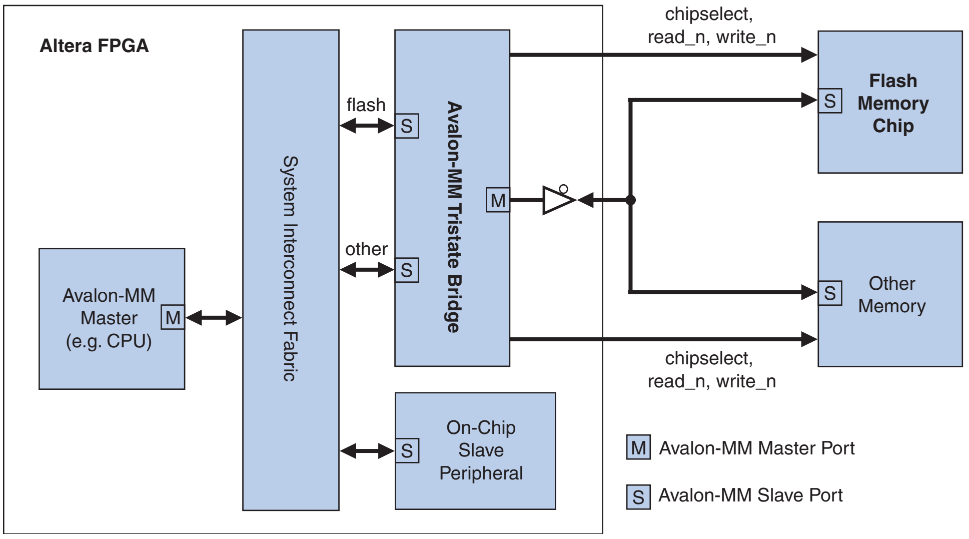
This chapter contains the following sections:

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


Altera FPGA
Avalon-MM Master (e.g. CPU)
M
System Interconnect Fabric
flash
S
other
S
Avalon-MM Tristate Bridge
M
S
On-Chip Slave Peripheral
chipselect, read_n, write_n
Flash Memory Chip
S
Other Memory
S
chipselect, read_n, write_n
M Avalon-MM Master Port
S Avalon-MM Slave Port

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 the Arria™ GX, Stratix® III, Stratix II GX, Stratix II, Stratix GX, Stratix, Cyclone® III, Cyclone II, and Cyclone device families. The CFI controller provides drivers for the Nios II HAL system library. No software support is provided for the first-generation Nios processor.

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 in the MegaWizard interface.

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.

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. Refer to the *Avalon Memory-Mapped Interface Specification* for details about dynamic bus sizing.

Timing Page

The options on this page specify the timing requirements for read and write transfers with the flash device. The settings available on the Timing page are:

- **Setup**—After asserting `chipselect`, the time required before asserting the `read` or `write` signals.
- **Wait**—The time required for the `read` or `write` signals to be asserted for each transfer.
- **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, us, ms, and clock cycles.

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

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 AMD and Intel 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 Memory-Mapped Interface Specification*
- *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
October 2007 v7.2.0	No change from previous release.	—
May 2007 v7.1.0	● Added Arria™ GX, Stratix II GX and Stratix GX to "Device and Tools Support" on page 3–2. ● Removed Board Info section from MegaWizard interface because it is no longer included with the device in 7.1. ● Added table of contents to Overview section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	Added Cyclone III support.	Version 7.0 of the Quartus II software added Cyclone III support.
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies ● Changed old "Avalon switch fabric" term to "system interconnect fabric" ● Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface" ● Added support for Stratix III devices	For the 6.1 release, added Stratix III device support. Additionally, Altera released the Avalon Streaming interface, which necessitated some rephrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
December 2004 v1.2	Added Cyclone II support.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

4. EPCS Device Controller Core

NII51012-7.2.0

Core Overview

The EPCS device 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 controller, Nios II systems can:

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

The EPCS controller 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) 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*.

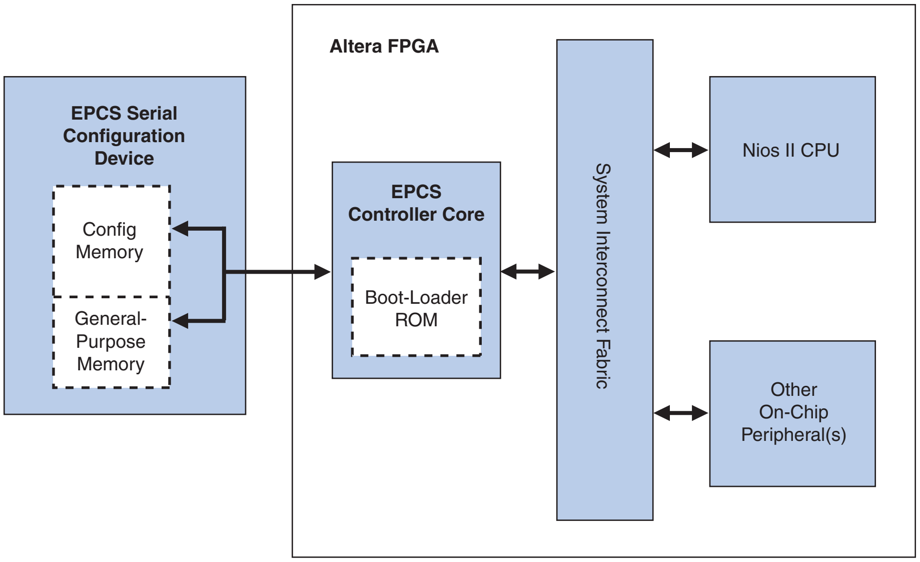
For Nios II processor users, the EPCS controller core supersedes the Active Serial Memory Interface (ASMI) device. New designs should use the EPCS controller instead of the ASMI core.

Functional Description

Figure 4–1 shows a block diagram of the EPCS controller 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 Controller



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 controller core contains an on-chip memory for storing a boot-loader program. When used in conjunction with Cyclone®, Cyclone II, and Cyclone III devices, the core requires 512 bytes of boot-loader ROM. For Stratix® II and Stratix III devices, the core requires 1 Kbyte of boot-loader ROM. The Nios II processor can be configured to boot from the EPCS controller. To do so, set the Nios II reset address to the base address of the EPCS controller. 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.

Refer to the *Nios II Flash Programmer User Guide*.

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, the EPCS 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 controller and the EPCS device. When you compile the SOPC Builder system in the Quartus II software, the EPCS controller core signals are routed automatically to the device pins for the EPCS device.

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.

In Cyclone III, the EPCS controller does not automatically assign its output pins to the dedicated configuration pins on the FPGA. Instead, the output pins are exported to the top level design, giving users the flexibility to connect to any EPCS devices. For more information on the configuration pins in Cyclone III, refer to the Pin-Out Files for Altera Device page.

Avalon-MM Slave Interface and Registers

The EPCS 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 on page 4–4, the first 256 words are dedicated to the boot-loader code, and the next seven words are control and data registers. A Nios II CPU can read 256 instruction words, starting from the EPCS controller's base address as flat memory space, which enables the CPU to reset into the EPCS controller's address space.

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

Table 4–1. EPCS Controller Register Map

Offset	Register Name	R/W	Bit Description
			31...0
0x000	Boot ROM Memory	R	Boot Loader Code
...			
0x0FF			
0x100	Read Data	R	(1)
0x101	Write Data	W	(1)
0x102	Status	R/W	(1)
0x103	Control	R/W	(1)
0x104	Reserved	—	(1)
0x105	Slave Enable	R/W	(1)
0x106	End of Packet	R/W	(1)

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 controller supports all Altera FPGA families that support the EPCS configuration device, such as the Cyclone device family. The EPCS controller 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. No software support is provided for any other processor, including the first-generation Nios.

Instantiating the Core in SOPC Builder

Hardware designers use the EPCS controller's SOPC Builder configuration wizard to add the EPCS controller to a system. There are no user-configurable settings for this component.

Only one EPCS controller can be instantiated in each FPGA design.

Software Programming Model

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

Document Revision History

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

Table 4–2. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	● Chapter 4 was formerly Chapter 3. ● Added sentence stating that to boot from EPCS controller memory set Nios II reset address to the base address of the EPCS controller. ● Added description on output pins assignment for Cyclone III in the Functional Description section.	—
May 2007 v7.1.0	● Removed text about reference designator from section on the configuration wizard because this setting is no longer available. ● Added sentence describing the purpose of the interrupt signal.	Version 7.1 updates text for changes in the parameter sheets and to clarify use of the interrupt signal.
March 2007 v7.0.0	Added Cyclone III support.	Version 7.0 of the Quartus II software added Cyclone III support.
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface” ● Changed old “Avalon switch fabric” term to “system interconnect fabric” ● Added ROM memory requirements for Cyclone, Cyclone II and Stratix II devices in section “Functional Description” on page 3–2 ● Added Stratix III device support	For the 6.1 release, added Stratix II support. Additionally, Altera released the Avalon Streaming interface, which necessitated some rephrasing of existing Avalon terminology. Other changes to the document serve only to clarify existing behavior.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

5. On-Chip FIFO Memory Core

QII55002-7.2.0

Core Overview

The on-chip FIFO memory core is a configurable component used to buffer data and provide flow control in an SOPC Builder system. The FIFO can operate with a single clock or with separate clocks for the input and output ports.

The input interface to the FIFO may be an Avalon® Memory Mapped (Avalon-MM) write slave or an Avalon Streaming (Avalon-ST) sink. The output interface can be a 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 includes an optional status interface that provides information about the fill-level of the FIFO. 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

The on-chip FIFO memory 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

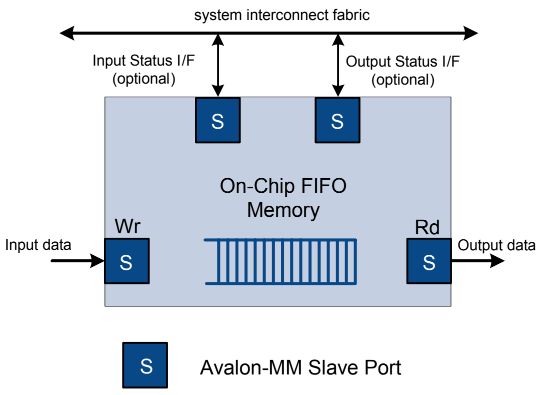
In all configurations, the input and output interfaces can use the optional backpressure signals to prevent underflow and overflow conditions. For the Avalon-MM interface, backpressure is implemented using the `waitrequest` signal. For Avalon-ST interfaces, backpressure is implemented using the `ready` and `valid` signals. For the on-chip FIFO memory, the delay between the sink asserts `ready` and the source drives valid data is one cycle. Bursting to a FIFO is not supported.

Avalon-MM Write Slave to Avalon-MM Read Slave

In this mode, the FIFO's input is a zero-address-width Avalon-MM write slave. An Avalon-MM write master pushes data into the FIFO by writing to the input interface, and a read master (possibly the same master) pops data by reading from its output interface. The FIFO's 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. `waitrequest` is only deasserted when there is enough space in the FIFO for a new transaction to complete. `waitrequest` is asserted for read operations when there is no data to be read from the FIFO, and is deasserted when the FIFO has data.

Figure 5–1. FIFO with Avalon-MM Input and Output Interfaces

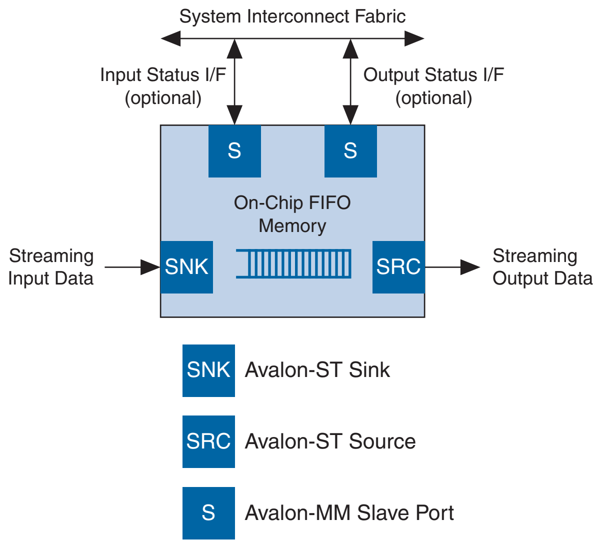

system interconnect fabric
Input Status I/F (optional)
Output Status I/F (optional)
S
S
On-Chip FIFO Memory
Wr
Rd
Input data
Output data
S
S
S
Avalon-MM Slave Port

Avalon-ST Sink to Avalon-ST Source

This FIFO has streaming input and output interfaces as illustrated in Figure 5–2. 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 on in the SOPC Builder MegaWizard, both interfaces use the `ready` and `valid` signals to indicate when space is available in the FIFO and when valid data is available.

For more information about the Avalon-ST interface protocol, refer to the *Avalon Streaming Interface Specification* available at **www.altera.com**.

Figure 5–2. FIFO with Avalon-ST Input and Output Interfaces

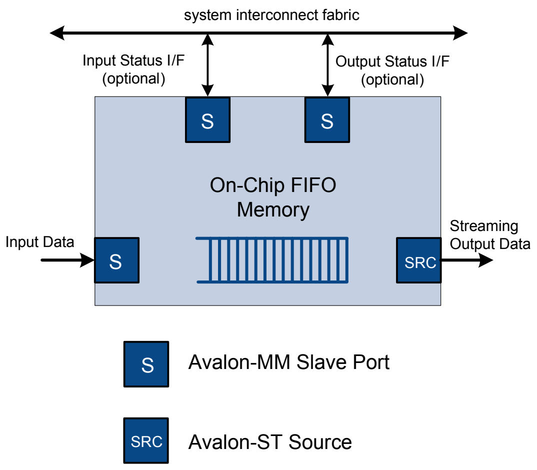


Avalon-MM Write Slave to Avalon-ST Source

In this mode, the FIFO's input is an Avalon-MM write slave with a width of 32 bits as shown in Figure 5–3. 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 performs the endian conversion to conform to the output interface protocol.

The signals that comprise this interface are mapped into bits in the Avalon's address space. If **Allow backpressure** is on, the input interface asserts `waitrequest` to indicate that the FIFO does not have enough space for the transaction to complete.

Figure 5–3. FIFO with Avalon-MM Input Interface and Avalon-ST Output Interface



The example memory map in Table 5–1 illustrates the layout of memory for a FIFO with a 32-bit Avalon-MM input interface and an Avalon-ST output interface. The output interface has 8-bit symbols, a 5-bit channel signal, and a 3-bit error signal, with packet support.

Table 5–1. Avalon-MM to Avalon-ST Memory Map

Offset	31 24	23 19	18 16	15 13	12 8	7 4	3 2	1	0
base + 0	Symbol 3	Symbol 2		Symbol 1		Symbol 0			
base + 1	reserved	reserved	error	resrvd.	channel	reserved	empty	EOP	SOP

If **Enable packet data** is off, the Avalon-MM write master writes all data at address offset 0 repeatedly to push data into the FIFO.

If **Enable packet data** is 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. The Avalon-MM master then writes packet data to the FIFO repeatedly at address offset 0, pushing 8-bit symbols into the FIFO. Whenever a valid write occurs at address offset 0, the data and its respective packet information is pushed into the FIFO. Subsequent data is written at address offset 0 without the need to clear the SOP. Rewriting to address offset 1 is not required each time if the subsequent data to be pushed into the FIFO is not the end-of-packet data, as long as `error` and `channel` do not change.

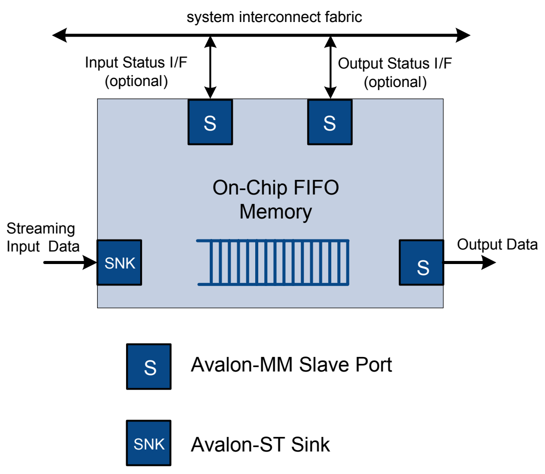
At the end of each packet, the Avalon-MM master writes to the address at offset 1 to set the EOP bit to 1, before writing the last symbol of the packet at offset 0. The write master uses the `empty` field to indicate the number of unused symbols at the end of the transfer. If the last packet data is not aligned with the symbols per beat, then 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, then 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 mode, the FIFO's input is an Avalon-ST sink and the output is an Avalon-MM read slave with a width of 32 bits (Figure 5–4). 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 performs the endian conversion to conform to the output interface protocol.

An Avalon-MM master reads the data from the FIFO. The signals are mapped into bits in the Avalon's address space. If **Allow backpressure** is on in the SOPC Builder MegaWizard, the input (sink) interface uses the `ready` and `valid` signals to indicate when space is available in the FIFO 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. It is deasserted when the FIFO has data to send.

Figure 5–4. FIFO with Avalon-ST Input and Avalon-MM Output



As shown in Table 5–2, the memory map for the Avalon-ST to Avalon-MM slave FIFO is exactly the same as for Avalon-MM to Avalon-ST FIFO.

Table 5–2. Avalon-ST to Avalon-MM Memory Map

Offset	31 … 24	23 … 19	18 … 16	15 … 13	12 … 8	7 … 4	3 … 2	1	0
base + 0	Symbol 3	Symbol 2		Symbol 1		Symbol 0			
base + 1	reserved	reserved	error	resrvd.	channel	reserved	empty	EOP	SOP

If **Enable packet data** is off, read data repeatedly at address offset 0 to pop the data from the FIFO.

If **Enable packet data** is on, the Avalon-MM read master starts reading from address offset 0. If the read is valid, that is, the FIFO is not empty, both data and packet status information are popped from the FIFO. The packet status information is obtained by reading at address offset 1. Reading from address offset 1 does not pop data from the FIFO. 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, then the `empty` field will be 3 to indicate that 3 symbols (the 3 least significant symbols in the memory map) are empty.

Status Interfaces

The FIFO 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 FIFOs that operate in a single domain, a single status interface is sufficient to monitor the status of the FIFO. For FIFOs using a dual clocking scheme, a second status interface using the output clock is necessary to accurately monitor the status of the FIFO in both clock domains.

Clocking Modes

When single clock mode is used, the FIFO being used is SCFIFO. When dual-clock mode is chosen, the FIFO 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 and Tools Support

The on-chip FIFO memory supports the Arria™ GX, Stratix® III, Stratix II GX, Stratix II, Stratix GX, Stratix, Cyclone® III, Cyclone II, Cyclone and Hardcopy® II device families.

Instantiating the Core in SOPC Builder

Designers use the MegaWizard® interface for the on-chip FIFO memory in SOPC Builder to specify the core configuration. The following sections describe the available options in the MegaWizard interface.

FIFO Settings

The following sections outline the settings that pertain to the FIFO as a whole.

Depth

Depth indicates the depth of the FIFO, 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 is built from registers or embedded memory blocks. The default is to construct the FIFO 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 will include the `waitrequest` signal which is asserted to prevent a master from writing to a full FIFO or reading from an empty FIFO. An Avalon-ST interface will include 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 cautious 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 5–3 lists all of the available functions.

Table 5–3. 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.
`altera_avalon_fifo_write_ienable()`	Writes the specified bits of the `interruptenable` register and performs error checking.

Table 5–3. On-Chip FIFO Memory Functions (Part 2 of 2)

Function Name	Description
`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 **Enable packet data** is 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 **Enable packet data** is on.

Software Control

Table 5–4 provides the register map for the `status` register. The layout of `status` register for the input and output interfaces is identical.

Table 5–4. 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						interruptenable					
base + 4	almostfull										
base + 5	almostempty										

Table 5–5 outlines the use of the various fields of the `status` register.

Table 5–5. 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 5–6 for the meaning of each bit field.

Table 5–5. FIFO Status Field Descriptions (Part 2 of 2)

Field	Type	Description
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.
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 will be used when attempting to write a value smaller than 1. The default will be 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 will be used when attempting to write a value smaller than 1. The maximum allowable will be used when attempting to write a value larger than the maximum allowable.

Table 5–6 describes the instantaneous status bits.

Table 5–6. Status Bit Field Descriptions

Bit(s)	Name	Description
1	FULL	Has a value of 1 if the FIFO is currently full.
0	EMPTY	Has a value of 1 if the FIFO is currently empty.
3	ALMOSTFULL	Has a value of 1 if the fill level of the FIFO is greater than the `almostfull` value.
2	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 5–7 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 5–7. 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 5–8 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 5–8. InterruptEnable Bit Field Descriptions (Part 1 of 2)

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.

Table 5–8. InterruptEnable Bit Field Descriptions (Part 2 of 2)

Bit(s)	Name	Description
5	IE_UNDERFLOW	Enables an interrupt is 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

An extensive programming example for the on-chip FIFO memory appears next to this document on the Quartus II literature page. Visit **www.altera.com/literature/quartus2/lit-qts-peripherals.jsp**.

Example 5–1. Sample Code for the On-Chip FIFO Memory

```
/***********************************************************************/
//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. */

    /* 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 `almostempy` 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 fill level of the FIFO.
Description:	Gets the fill level of the FIFO which is 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 don't)
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 t o Avalon-MM transfers. See the *Avalon Memory-Mapped and Avalon Streaming 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 Memory-Mapped and Avalon Streaming 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:	Retunrs the packet status information from address offset 1 of the Avalon interface. See the *Avalon Memory-Mapped and Avalon Streaming 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 the Avalon Streaming Interface Specification.

Document Revision History

Table 5–9 shows the revision history for this chapter.

Table 5–9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 5 was formerly Chapter 4.	—
May 2007 v7.1.0	Initial release.	—

6. Scatter-Gather DMA Controller Core

QII55003-7.2.1

Core Overview

The Scatter-Gather direct memory access (SG-DMA) controller core implements high-speed data transfer between two devices. The SG-DMA core can be used 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. It also transfers contiguous memory to non-contiguous memory. The core operates by reading 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 HAL system library, allowing software to access the core using the ANSI C Standard Library **stdio.h** routines.

Example Systems

Figure 6–1 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. The descriptor table, containing the linked list of descriptors specifying data transfers to be executed, can be located in the FPGA or an external memory. Locating this table in an external memory will free up resources in the FPGA; however, an external descriptor table will increase the overhead involved when the descriptor processor reads and updates the table. The SG-DMA core has an internal FIFO to store descriptors read from memory, which allows it to perform descriptor read, execute, and write back operations in parallel, hiding the descriptor read latency.

Figure 6–1. Scatter-Gather DMA Controller Core with Streaming Peripheral and External Memory

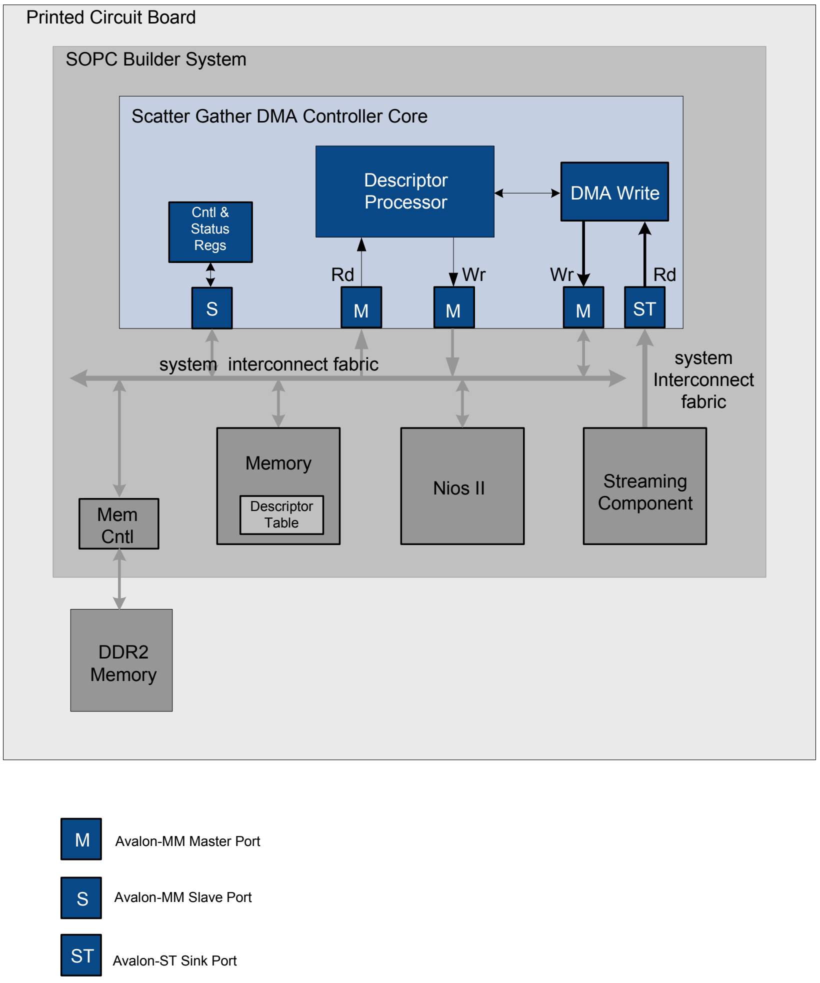
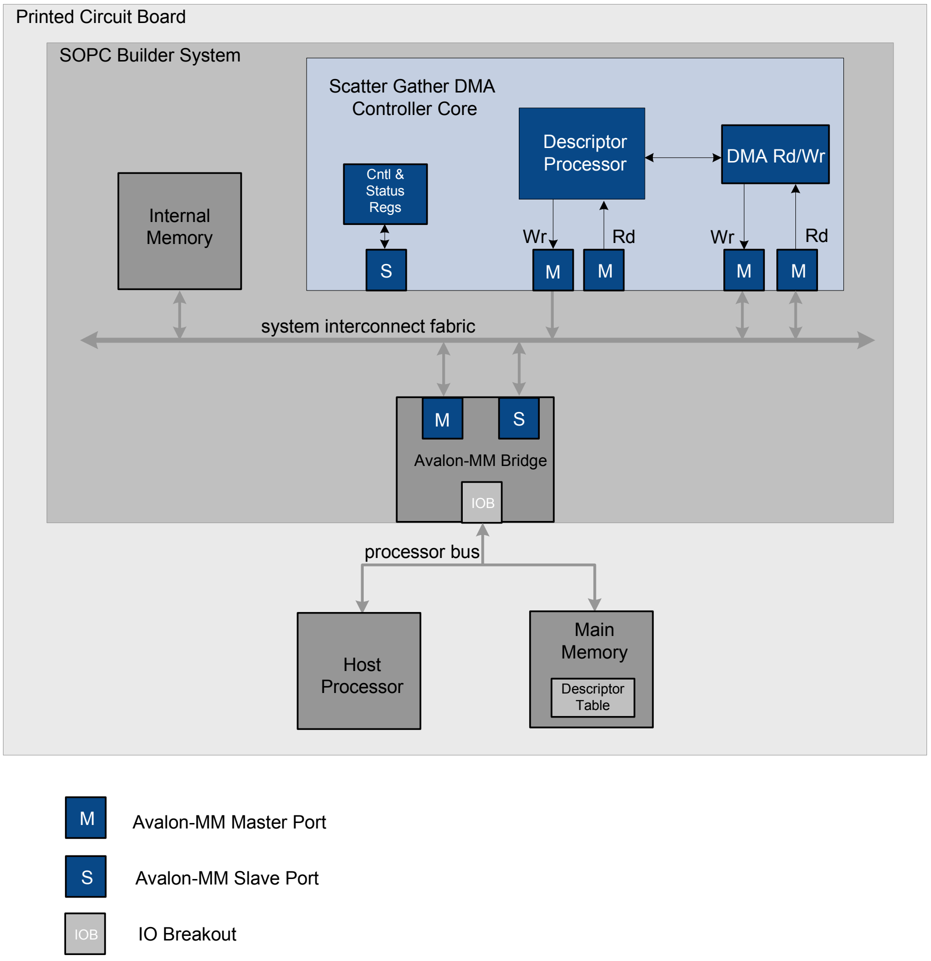


Figure 6–2 shows a different use of the SG-DMA controller core. In Figure 6–2, SG-DMA core transfers data between an internal and external memory. The host processor and memory are on the system bus, typically either a PCI-Express or Serial RapidIO™.

Figure 6–2. Scatter-Gather DMA Controller Core with Internal and External Memory



Figures 6–1 and 6–2 illustrate systems using the SG-DMA controller core and omit some of the internals of the core itself. Figure 6–3 on page 6–7 illustrates all of the SG-DMA controller core internals.

Resource Usage and Performance

Resource utilization for the core is 600–1400 LEs, depending upon the width of the datapath, the parameterization of the core, and the type of data transfer. Table 6–1 provides resource utilization for a SG-DMA controller core used for memory to memory transfer. The core is highly parameterized and the resource utilization will vary with the configuration specified.

Table 6–1. SG-DMA Estimated Resource Usage

Datapath	Cyclone® II	Stratix® (Approx. LEs)	Stratix II (Approx. 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 6–2 provides an example of expected performance for SG-DMA cores instantiated in several different device families.

Table 6–2. SG-DMA Performance Estimates

Device	Datapath	Fmax	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

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 perform a single DMA transfer at a time. With the older DMA controller core, a CPU performed separate reads for each entry of the DMA descriptor table and then executed separate IO writes to program the DMA controller to perform the transfer. 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.

Functional Description

The SG-DMA controller core comprises three major blocks: a descriptor processor, a DMA read block and a DMA write block. (See Figure 6–3 on page 6–7.) These blocks are combined to create three different configurations:

- memory to memory
- memory to stream
- stream to memory

For the memory-to-memory configuration, the core includes all three blocks. If the core is configured for memory-to-stream transactions, only the descriptor processor and read blocks are required. If the core is configured for stream-to-memory transactions, only the descriptor processor and write blocks are required. In the memory-to-memory configuration, an internal FIFO holds data being transferred between the read and write blocks. In the other two configurations, an external FIFO might be required depending upon the throughput of the components being connected. For designs requiring an external FIFO, the on-chip FIFO memory available in SOPC Builder can be used.

The following sections describe the three configurations of the SG-DMA controller core and the behavior of the internal modules for each configuration.

Memory-to-Memory Configuration

As Figure 6–3 illustrates, the memory-to-memory configuration includes five Avalon-MM ports. The descriptor processor block uses read and write Avalon-MM master ports to access and update descriptors. The DMA read block has an Avalon-MM master read port to read data from memory; it has an Avalon-ST port to pass data to the DMA write block. The DMA write block has an Avalon-ST port to receive data from the read block and an Avalon-MM master write port to write the data to memory. Software accesses an Avalon-MM slave port to read and write the `control` and `status` registers.

In the memory-to-memory configuration, the descriptor processor reads descriptors from the descriptor table and pushes the appropriate commands onto the input FIFOs of the DMA read and write blocks. It also receives a *status token* from the read or write block after each descriptor has been processed. The status token contains information about the status of the transfer, including the number of bytes transferred. The descriptor processor then writes this information back to the appropriate entry in the descriptor table.

In the memory-to-memory configuration, an internal data FIFO stores data being transferred between the read and write blocks to provide buffering and flow control.

To execute a DMA read transfer between memories, the following steps are executed:

1. Software writes the descriptor into memory.

2. Software writes the location of the first descriptor address to SG-DMA controller hardware and initiates the transfer by setting the RUN bit of the SG-DMA `control` register.

3. The descriptor processor reads the descriptors from memory and writes them into a command FIFO which feeds commands to both the DMA read and write blocks.

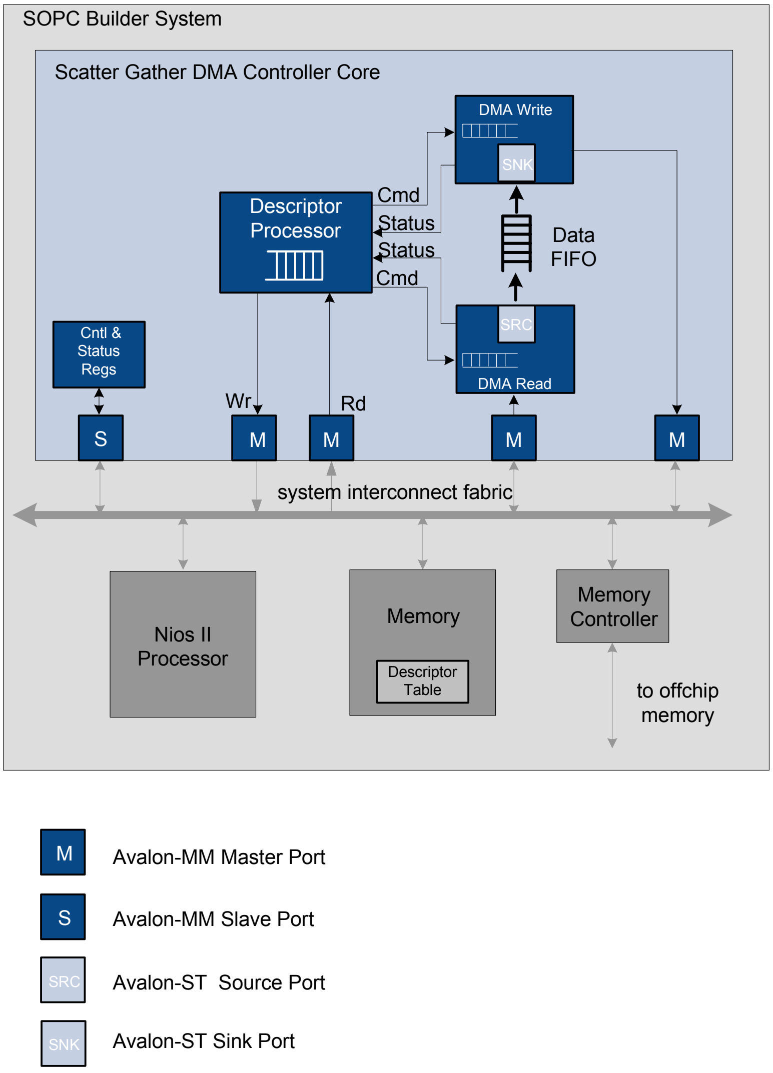
4. The DMA read block gets the source address from its command FIFO and reads data to fill the FIFO on its stream port. The read block continues reading until the specified number of bytes have been transferred. If the data FIFO ever fills, the read block pauses until the FIFO can accept more data.

5. The DMA write block gets the destination address from its command FIFO. The write block continues to execute writes until the specified number of bytes have been transferred. It then sends a status update to the DMA controller. If the data FIFO ever empties, the write block pauses until the FIFO has more data to write.

6. The descriptor processor updates the appropriate entry in the descriptor table.

Figure 6–3 illustrates one possible configuration for the memory-to-memory SG-DMA controller with an internal Nios II processor and descriptor table.

Figure 6–3. Scatter-Gather DMA Controller Core for Memory-to-Memory Configuration

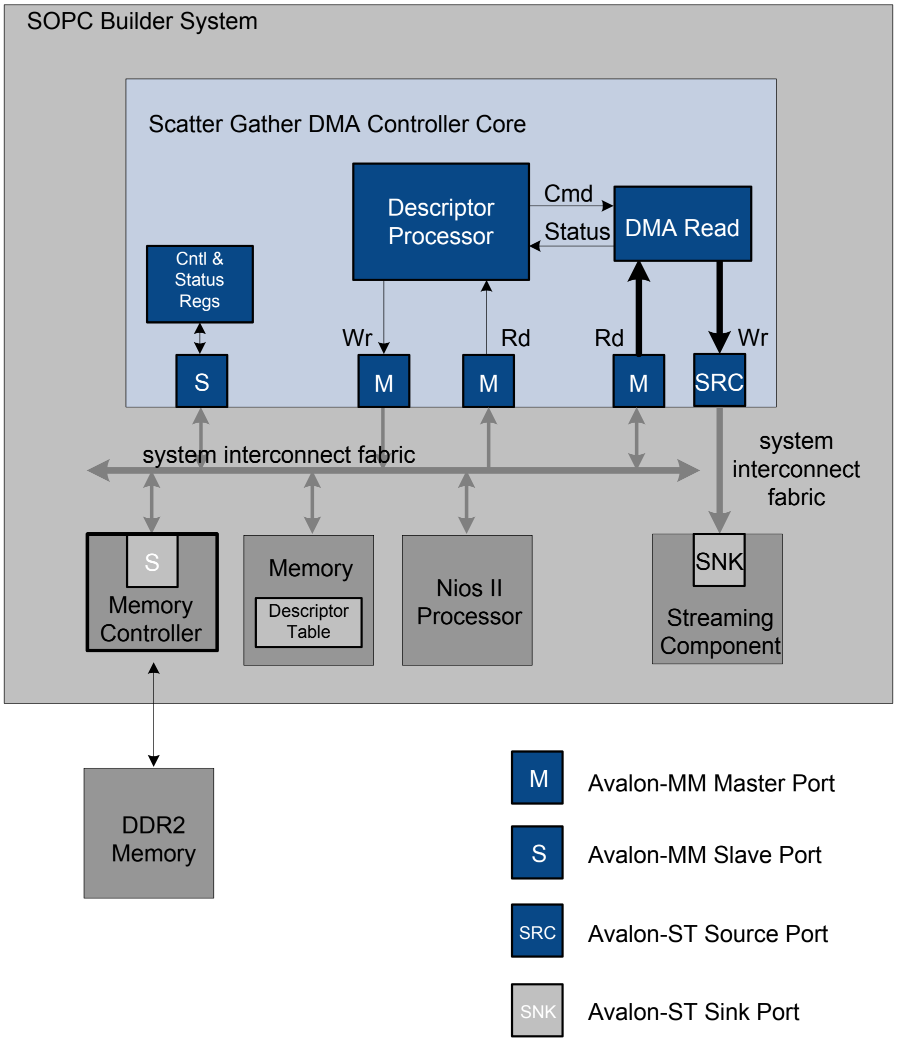


Memory-to-Stream Configuration

The memory-to-stream configuration includes the descriptor processor and the DMA read block. As Figure 6–3 illustrates, this configuration includes four Avalon-MM ports and one Avalon-ST port. The descriptor processor block includes read and write Avalon-MM master ports to access and update descriptors. The DMA read block has an Avalon-MM master read port to read data from memory and an Avalon-ST port to write the data to a streaming component. An Avalon-MM slave port is used to read and write the `control` and `status` registers.

Figure 6–4 on page 6–9 illustrates a SG-DMA controller in the memory-to-stream configuration. In this example, the Nios II processor and descriptor table are inside the FPGA. Data from an external DDR2 memory is read by the SG-DMA controller and written to an internal streaming peripheral. The read block returns status to the descriptor processor upon completion of each descriptor.

Figure 6–4. Scatter-Gather DMA Controller Memory-to-Stream Configuration



The transfer operation includes the following steps:

1. Software writes the descriptor into memory.

2. Software writes the location of first descriptor address to SG-DMA controller hardware and initiates the transfer by setting the RUN bit of the SG-DMA `control` register.

3. The descriptor processor reads the descriptors from memory and writes them into the input command FIFO in the read block.

4. The read block reads from the source address and pushes the data to its stream port. The read block continues reads until the specified number of bytes have been transferred. It then sends a status update to the descriptor processor.

5. The descriptor processor updates the appropriate entry in the descriptor table.

Stream-to-Memory Configuration

The stream-to-memory configuration includes the descriptor processor and the DMA write block. The write block returns status to the descriptor processor upon completion of each descriptor. This configuration is similar to the memory-to-stream configuration except that the data flows from a streaming component to a memory device as Figure 6–5 illustrates. In this example, an On-Chip FIFO Memory component is used to provide a buffer between the streaming component and the DMA write.

The transfer operation includes the following steps:

1. Software writes the descriptor into memory.

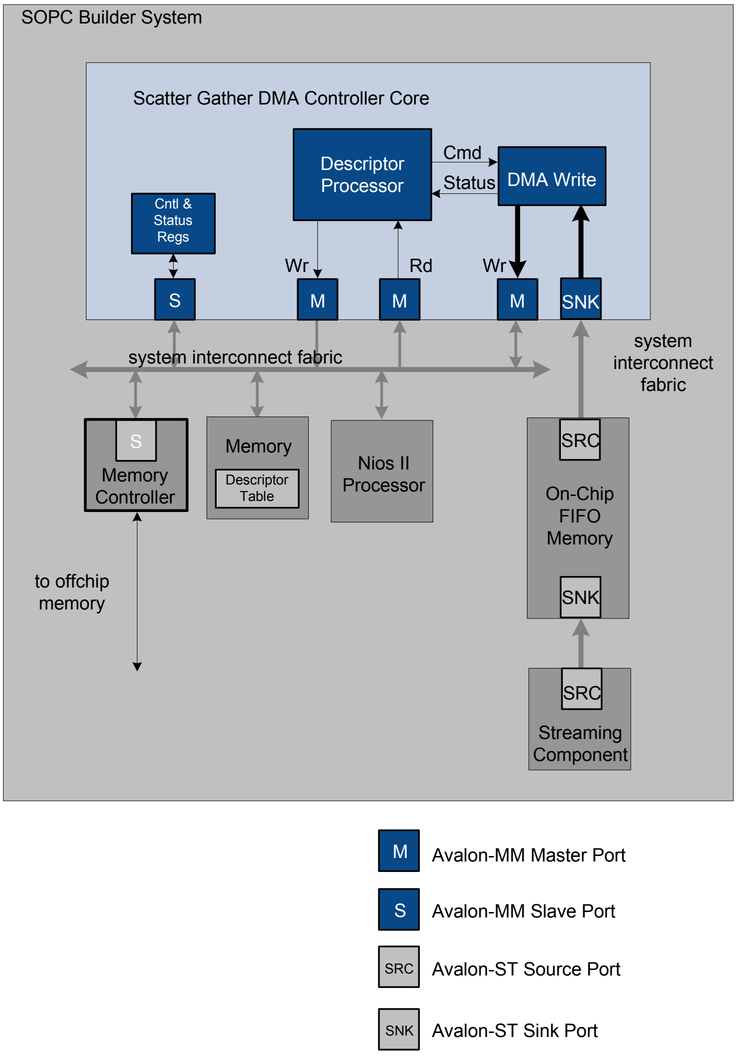
2. Software writes the location of the first descriptor address to SG-DMA controller hardware and initiates the transfer by writing the RUN bit of the SG-DMA `control` register.

3. The descriptor processor reads the descriptors from memory and writes them into the write block.

4. The write block reads from its stream port and writes the data to its Avalon master port. The write block continues reads until the specified number of bytes have been transferred. It then sends a status update to the descriptor processor.

5. The descriptor processor updates the appropriate entry in the descriptor table.

Figure 6–5. Scatter-Gather DMA Controller Stream-to-Memory Configuration



Possible Sources of Errors

The SG-DMA core has a parameterizable error width. Error signals are wired directly to the Avalon-ST source or sink to which the SG-DMA core is connected. Table 6–3 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®.

Table 6–3. Avalon-ST Transmit Channel 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 6–4 lists the error signals when the core is operating in the stream-to-memory configuration and connected to the transmit FIFO interface of the Altera Triple-Speed Ethernet MegaCore.

Table 6–4. Avalon-ST Receive Channel 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.

Detailed Description of Each Block

The following sections provide a detailed description of each functional block.

Descriptor Processor Block

The descriptor processor reads descriptors from memory using its Avalon-MM descriptor read master port and pushes commands onto the command FIFOs of the DMA read and write blocks. The command includes the following fields to specify the transfers:

- source address
- destination address
- read size
- write size
- bytes to transfer
- increment read address after each transfer
- increment write address after each transfer
- generate end of packet

DMA Read Block

The DMA read block reads commands from its input command FIFO. For each command, it reads data from the source address on its Avalon-MM port. In the memory-to-memory configuration, it pushes the data into the data FIFO. In the memory-to-stream configuration, it immediately writes the data to the Avalon-ST source port.

☞ The DMA read block will not begin an Avalon-MM read unless its data FIFO has enough space to store all of the data read. This restriction requires the external FIFO be at least as deep as the maximum supported read size.

DMA Write Block

The DMA write block reads commands from its input command FIFO. For each command, it writes data received on its Avalon-ST sink port to the destination address. In the memory-to-memory and the stream-to-memory configurations, it reads the data from its Avalon-ST port and writes to its Avalon-MM port.

Device Support and Tools

The SG-DMA controller core supports the Arria™ GX, Stratix® III, Stratix II GX, Stratix II, Stratix, Cyclone® III, Cyclone II, Cyclone and Hardcopy® II device families.

Instantiating the Core in SOPC Builder

Hardware designers use the MegaWizard interface for the SG-DMA controller core in SOPC Builder to specify the core configuration.

☞ If an SOPC Builder system contains both the SG-DMA controller and JTAG UART cores, set the IRQs for the SG-DMA controller core to a higher priority than the IRQs for the JTAG UART core.

The following sections describe the available options in the configuration wizard.

Transfer Mode

This list allows you to select between the **Memory To Memory**, **Memory To Stream**, and **Stream To Memory** configurations. For more information about these configurations, see "Memory-to-Memory Configuration" on page 6–5, "Memory-to-Stream Configuration" on page 6–7, and "Stream-to-Memory Configuration" on page 6–10.

Allow Unaligned Transfers

If **Allow unaligned transfers** is on, data transfers for data widths that are not a power of two will be aligned on word boundaries. Unaligned transfers require extra logic that may negatively impact system performance.

Data and Error Widths

The **Data width** list allows you to select data width in bits for the Avalon-MM read and write ports. The **Source error width** and **Sink error width** lists allow you to select widths of the error signals for the Avalon-ST source and sink ports.

FIFO Depth

The **Data transfer FIFO depth** list sets the depth for all three descriptor FIFOs:

- the Descriptor Processor block FIFO
- the DMA read block FIFO
- the DMA write block FIFO

The **Data transfer FIFO depth** list also sets the depth for the internal data FIFO used in the memory-to-memory configuration. These FIFOs are all illustrated in Figure 6–3 on page 6–7.

Hardware Simulation Considerations

Signals for hardware simulation are automatically generated and show up as part of the Nios II simulation accessible from the Nios II IDE. On the Run menu, point to Run As and click **Nios II Modelsim**.

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 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.

Programming with the SG-DMA Controller

This section describes the software constructs for programming the SG-DMA Controller.

Table 6–5. SG-DMA Controller Functions

Function Name	Description
`alt_avalon_sgdma_do_async_transfer()`	Sets up and begins a non-blocking transfer of one or more descriptors or a descriptor chain.
`alt_avalon_sgdma_do_sync_transfer()`	Sends a fully formed descriptor, or list of descriptors, to the SG-DMA Controller for transfer. This function will block 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.
`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` register of the descriptor.
`alt_avalon_sgdma_register_callback()`	Associates a user-specific routine with the SG-DMA interrupt handler.
`alt_avalon_sgdma_start()`	Starts the DMA engine.
`alt_avalon_sgdma_stop()`	Stops the DMA engine.
`alt_avalon_sgdma_open()`	Retrieves a pointer to the SG-DMA controller with the given name.

Software Control

The host processor programs the SG-DMA by writing to its `control` register. The host processor reads SG-DMA `status` register to determine the current status.

Table 6–6 shows the offsets for the `control` and `status` registers.

Table 6–6. SG-DMA Control and Status Slave Register Map

Offset	Register Name
base + 0	`status`
base + 1	`control`
base + 2	`next_descriptor_pointer`

Software writes the `control` register to specify the behavior of the SG-DMA controller. This register determines the conditions under which the SG-DMA controller will generate an interrupt. It also includes the control bits used to start and stop processing descriptors.

☞ Bursting to the `control` port is not supported.

Table 6–7 provides a bit-map for the `control` register.

Table 6–7. SG-DMA Control Register Map (Part 1 of 2)

Bit	Bit Name	Rd/Wr/Clr	Description
0	`IE_ERROR`	R/W	Assert an interrupt when (`ERROR` = 1).
1	`IE_EOP_ENCOUNTERED`	R/W	Assert an interrupt when (`EOP_ENCOUNTERED = 1`).
2	`IE_DESCRIPTOR_COMPLETED`	R/W	Assert an interrupt when (`DESCRIPTOR_COMPLETED = 1`).
3	`IE_CHAIN_COMPLETED`	R/W	Assert an interrupt when (`CHAIN COMPLETED =1`).
4	`IE_GLOBAL`	R/W	Global signal to enable all interrupts.
5	`RUN`	R/W	The SG-DMA processes descriptors in its queue as long as `RUN = 1`. The SG-DMA will not process the next descriptor in its queue when `RUN = 0`. Setting RUN starts the descriptor processor that initiates DMA transactions. Clearing `RUN` will not stop processing of a descriptor if processing has already begun.
6	`STOP_DMA_ER`	R/W	Stops DMA after current descriptor if `ERROR` is detected.
7	`IE_MAX_DESC_PROCESSED` *(1)*	R/W	Enables interrupts when `MAX_DESC_PROCESSED` is reached.
8 .. 15	`MAX_DESC_PROCESSED` *(1)*	R/W	Specifies the number of descriptors to process before invoking interrupt.

Table 6–7. SG-DMA Control Register Map (Part 2 of 2)

Bit	Bit Name	Rd/Wr/Clr	Description
16	SW_RESET	R/W	Resets the SG-DMA hardware and stops all operations immediately.
17	PARK	R/W	Enables the hardware to repeatedly use the same descriptor without software intervention. The bit owned_by_hw is not cleared, thus allowing the hardware to reuse the descriptor.
18..30	Reserved		
31	CLEAR INTERRUPT	R/W	Set this bit to 1 to clear pending interrupts.

Note to Table 6–7:
(1) Available if interrupt coalescing is selected in the synthesis options.

Table 6–8 provides a bit-map for the status register.

Table 6–8. SG-DMA Status Register Map

Bit	Bit Name	Rd/Wr/Clr	Description
0	ERROR	R/C (1)(2)	Avalon-ST error encountered during transfer.
1	EOP_ENCOUNTERED	R/C	Transfer terminated by Avalon-ST EOP.
2	DESCRIPTOR_COMPLETED	R/C (1)(2)	A descriptor was processed to completion.
3	CHAIN_COMPLETED	R/C (1)(2)	Unable to process next descriptor because Owned by HW = 0.
4	BUSY	R/C (1)(3))	Indicates that descriptors are being processed; the linked list of descriptors is not yet completed.
5 .. 31	reserved		

Notes to Table 6–8:
(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.

Next Descriptor Pointer

Software writes the address of the first descriptor to this register as part of the system initialization sequence. When RUN = 1, the SG-DMA updates this register with the location of the next descriptor to be fetched. To stop execution of the SG-DMA core, software clears the RUN bit. When RUN is 0, the SG-DMA hardware completes the data transfers for the current descriptor and then stops processing. Software can then modify the remaining linked-list and restart the SG-DMA hardware.

☞ While `BUSY = 1`, the next descriptor pointer is updated by hardware. The next descriptor pointer can only be reliably read by software when `BUSY = 0`.

DMA Descriptors

The DMA descriptors specify all information required to perform data transfers, including: the source address, destination address, and the number of bytes to be transferred. The descriptors are stored in a table that is written by software. This table can be stored in FPGA memory or an external memory device as a linked list. The descriptor must be initialized by user applications and aligned on a 256-bit boundary.

Table 6–9 shows the layout of a descriptor entry.

Table 6–9. Descriptor Layout

Offset	Bit Field Names			
	31 24	23 16	15 8	7 0
base	SOURCE			
base + 1	RESERVED			
base + 2	DESTINATION			
base + 3	RESERVED			
base + 4	NEXT_DESC_PTR			
base + 5	RESERVED			
base + 6	WRITE_SIZE	READ_SIZE	BYTES_TO_TRANSFER	
base + 7	DESC_CONTROL	DESC_STATUS	ACTUAL_BYTES_TRANSFERRED	

Table 6–10 describes the function of the various fields.

Table 6–10. Descriptor Field Descriptions (Part 1 of 2)

Field Name	Rd/Wr/Clr	Description
SOURCE	R/W	Specifies the address of data to be read. This address is set to 0 if the input source is an Avalon Streaming (Avalon-ST) interface.
DESTINATION	R/W	Specifies the address to which data should be written. This address is set to 0 if the write data is an Avalon-ST interface.
NEXT_DESC_PTR	R/W	Specifies the next descriptor in the linked list.

Table 6–10. Descriptor Field Descriptions (Part 2 of 2)

Field Name	Rd/Wr/Clr	Description
`BYTES_TO_TRANSFER`	R/W	Specifies the number of bytes to transfer. If `BYTES_TO_TRANSFER = 0`, the transaction will be terminated by an EOP.
`READ_SIZE`	R/W	Specifies the read size in bytes for a read from Avalon devices (memory).
`WRITE_SIZE`	R/W	Specifies the write size in bytes for a write to Avalon Devices (memory).
`ACTUAL_BYTES_TRANSFERRED`	R	Specifies the number of bytes that are successfully transferred by the DMA hardware.
`DESC_CONTROL`	R/W	See Table 6–12 for descriptions of each bit.
`DESC_STATUS`	R/W	See Table 6–11 for descriptions of each bit.

The descriptor processor reads the `DESC_CONTROL` fields to determine how to proceed with the DMA transaction. Table 6–11 provides a bit-map for theses fields.

Table 6–11. Desc_Control Field Map

Bits	Field Name	Rd/Wr/Clr	Description
0	Generate_EOP	W	When set, DMA Read should generate an EOP on the final word.
1	Read_Fixed_Address	R/W	For Avalon-MM ports, when set to 1, DMA Read does not increment the memory address. When 0, the read address increments after each read. When used in Memory-to-Stream mode, the read engine generates a `startofpacket` signal on the first word.
2	Write_Fixed_Address	R/W	Used only for Avalon-MM ports. When set to 1, DMA Write does not increment the memory address. When 0, the write address increments after each write.

Table 6–11. Desc_Control Field Map

Bits	Field Name	Rd/Wr/Clr	Description
3 .. 6	Avalon-ST_Channel_Number	R/W	DMA Read drives this value onto the Avalon-ST channel port for each word in the transaction. The DMA Write replaces this value with the Avalon-ST channel number for its sink port.
7	Owned_by_HW	R/W	This bit determines whether hardware or software has write access to the descriptor of the `SG-DMA control` and `status` register. When `Owned_by_HW=1` the hardware can update this pointer. When `Owned_by_HW=0`, software can update this pointer.

After completing a DMA transaction, the descriptor processor updates the `DESC_STATUS` fields to indicate how the transaction proceeded. The error conditions these fields record can only occur on an Avalon-ST interface. Table 6–12 provides a bit-map for the `DESC_STATUS` fields.

Table 6–12. Descriptor Desc_Status Bit Map

Bit	Bit Name	Rd/Wr/Clr	Description
0	E_CRC	R	When set, indicates that a CRC error occurred on the Avalon-ST interface.
1	E_PARITY	R	When set, indicates that a parity error occurred on the Avalon-ST interface.
2	E_OVERFLOW	R	When set, indicates that an overflow occurred on the Avalon-ST interface.
3	E_SYNC	R	When set, indicates that an out-of-sync error occurred on the Avalon-ST interface.
4	E_UEOP	R	When set, indicates that an unexpected EOP error occurred on the Avalon-ST interface.
5	E_MEOP	R	When set, indicates that a missing EOP error occurred on the Avalon-ST interface.
6	E_MSOP	R	When set, indicates that a missing SOP error occurred on the Avalon-ST interface.
7	Terminated_by_ EOP	R	When set, indicates that a write transaction was terminated by EOP.

Macros to access all of the registers are defined in **altera_avalon_sgdma_regs.h.** For example, this file includes macros to access the `status` register, including the following macros:

```
#define IOADDR_ ALTERA_AVALON_SGDMA_STATUS(base)        __IO_CALC_ADDRESS_DYNAMIC(base, 0)
```

```
#define IORD_ALTERA_AVALON_SGDMA_STATUS(base)          IORD(base, 0)
#define IOWR_ALTERA_AVALON_SGDMA_STATUS(base, data)    IOWR(base, 0, data)
#define ALTERA_AVALON_SGDMA_STATUS_ERROR_MSK (0x1)
#define ALTERA_AVALON_SGDMA_STATUS_ERROR_OFST (0)
#define ALTERA_AVALON_SGDMA_STATUS_EOP_ENCOUNTERED_MSK (0x2)
#define ALTERA_AVALON_SGDMA_STATUS_EOP_ENCOUNTERED_OFST (1)
```

For a complete list of predefined macros and utilities to access the SG-DMA Controller hardware, see:

- **<*install_dir*>\quartus\sopc_builder\components\altera_avalon_sgdma\inc\altera_avalon_sgdma_regs.h**,
- **<*install_dir*>\quartus\sopc_builder\components\altera_avalon_sgdma\HAL\inc\altera_avalon_sgdma.h**, and
- **<*install_dir*>\quartus\sopc_builder\components\altera_avalon_sgdma\HAL\inc\altera_avalon_sgdma_descriptor.h.**

Timeouts

The SG-DMA controller does not implement internal counters to detect stalls. Software can instantiate a timer component if this functionality is required.

SG-DMA Controller API

This section describes the application programming interface (API) for the SG-DMA controller core.

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 will immediately return -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 will be set up to issue an interrupt on error, EOP, or chain completion. Otherwise, no interrupt is registered and it is the responsibility of the application developer to check for and handle errors and completion.

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. It is the responsibility of the user application to search through the descriptor or list of descriptors to gather specific error information.

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 will read from a fixed address.
`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-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 of the SG-DMA `control` register is asserted.

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 will 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_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 will continue 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 of the SG-DMA `control` register is asserted. 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 will 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_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 will read from a fixed address.
`generate_sop`—if non-zero, the SG-DMA will generate a start-of-packet (SOP) on the Avalon Streaming interface when commencing the transfer.
`generate_eop`—if non-zero, the SG-DMA will generate a end-of-packet (EOP) on the Avalon Streaming interface when completing the transfer.
`atlantic_channel`—an 8-bit channel identification number that will be passed to the Avalon-ST interface.

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 of the SG-DMA `control` register is asserted.

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 will 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 will enable interrupts that will cause the callback to be executed. The callback runs as part of the interrupt service routine, and great 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.

Document Revision History

Table 6–13 shows the revision history for this chapter.

Table 6–13. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
January 2008 v7.2.1	Updated Table 6–10.	Updated description of field names `READ_` and `WRITE_` in Table 6–10.
October 2007 v7.2.0	● Chapter 6 was formerly Chapter 5. ● Updated the description for the following sections and APIs: Instantiating the Core in SOPC Builder, Software Control, DMA Descriptors, alt_avalon_sgdma_start() and alt_avalon_sgdma_stop()	—
May 2007 v7.1.0	Initial release.	—

7. DMA Controller Core

NII51006-7.2.0

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-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, a universal asynchronous receiver/transmitter [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 7–6 for details of HAL support.

This chapter contains the following sections:

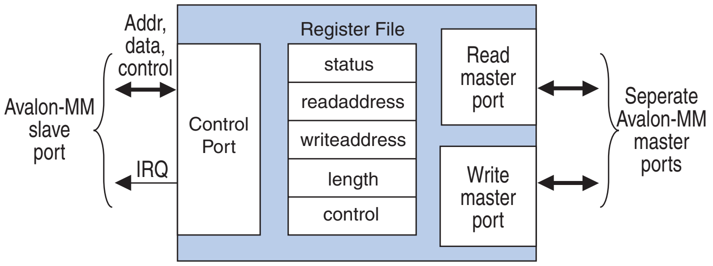
- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 7–4
- "Device and Tools Support" on page 7–6
- "Software Programming Model" on page 7–6

Functional Description

The DMA controller is used to perform direct memory-access data transfers from a source address-space to a destination address-space. The source and destination may be either an Avalon-MM slave peripheral (i.e., 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 7–1.

Figure 7–1. 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 (i.e., a fixed-length transaction), or an end-of-packet signal is asserted by either the sender or receiver (in other words, 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. The DMA controller is programmed 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 are capable of performing 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 the *Avalon Memory-Mapped Interface Specification*.

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 7–1.

Table 7–1. 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 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 has a strong impact on logic utilization when the DMA controller's data width is large. See "Advanced Options" on page 7–6.

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 amount 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 will only transfer data to the 16-bit memory, then 32-bit transfers could be disabled to conserve logic resources.

Device and Tools Support

The DMA Controller Core with Avalon Interface supports all Altera FPGA 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 will interfere 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 7–2 lists the available operations. These are valid for both the transmit and receive channels.

Table 7–2. 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 value of "arg" is ignored.
`ALT_DMA_SET_MODE_16`	Transfers data in units of 16 bits. The value of "arg" is ignored.
`ALT_DMA_SET_MODE_32`	Transfers data in units of 32 bits. The value of "arg" is ignored.
`ALT_DMA_SET_MODE_64`	Transfers data in units of 64 bits. The value of "arg" is ignored.
`ALT_DMA_SET_MODE_128`	Transfers data in units of 128 bits. The value of "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 "arg" parameter specifies the address to read from.
`ALT_DMA_RX_ONLY_OFF` *(1)*	Turns off streaming mode for a receive channel. The value of "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 "arg" parameter specifies the address to write to.
`ALT_DMA_TX_ONLY_OFF` *(1)*	Turns off streaming mode for a transmit channel. The value of "arg" is ignored.

Note to Table 7–2:

(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 7–3 shows the register map for the DMA controller. Device drivers control and communicate with the hardware through five memory-mapped 32-bit registers.

Table 7–3. 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 7–3:
(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 7–4.

Table 7–4. status Register Bits

Bit Number	Bit Name	Read/Write/Clear	Description
0	DONE	R/C	A DMA transaction is completed. 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.
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 7–5.

Table 7–5. 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.
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.

Table 7–5. control Register Bits (Part 2 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
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 7–4.
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 7–4.
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 will be 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 Document

This chapter references the Avalon Memory-Mapped Interface Specification manual.

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
October 2007 v7.2.0	● Chapter 7 was formerly Chapter 6. ● Updated the description on Burst Transactions parameters.	—
May 2007 v7.1.0	● Chapter 6 was formerly Chapter 4. ● Added "Device and Tools Support" on page 7–6 section. ● Added note on addressing native-aligned peripherals. ● Added table of contents to Overview section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies. Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface." ● Added description of `SOFTWARERESET` bit to `control` register in Table 4–5 on page 4–10. ● Added more information about DMA addressing and the fact that addresses are aligned to the size of the data transfer in "The Master Read and Write Ports" on page 4–3.	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology. Other changes to the document serve only to clarify existing behavior.
May 2006 v6.0.0	Chapter title changed, but no change in content from previous release.	—
December 2005 v5.1.1	Changed Avalon "streaming" terminology to "flow control" based on a change to the *Avalon Interface Specification*	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
December 2004 v1.2	● Updated description of the `GO` bit. ● Updated descriptions of `ioctl()` macros in table 6-2.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

Section II. Communication Peripherals

This section describes communication peripherals provided by Altera. These components provide communication interfaces for SOPC Builder systems.

See *About This Handbook* for further details.

This section includes the following chapters:

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

8. JTAG UART Core

NII51009-7.2.0

Core Overview

The JTAG universal asynchronous receiver/transmitter (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 a simple register-mapped 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.

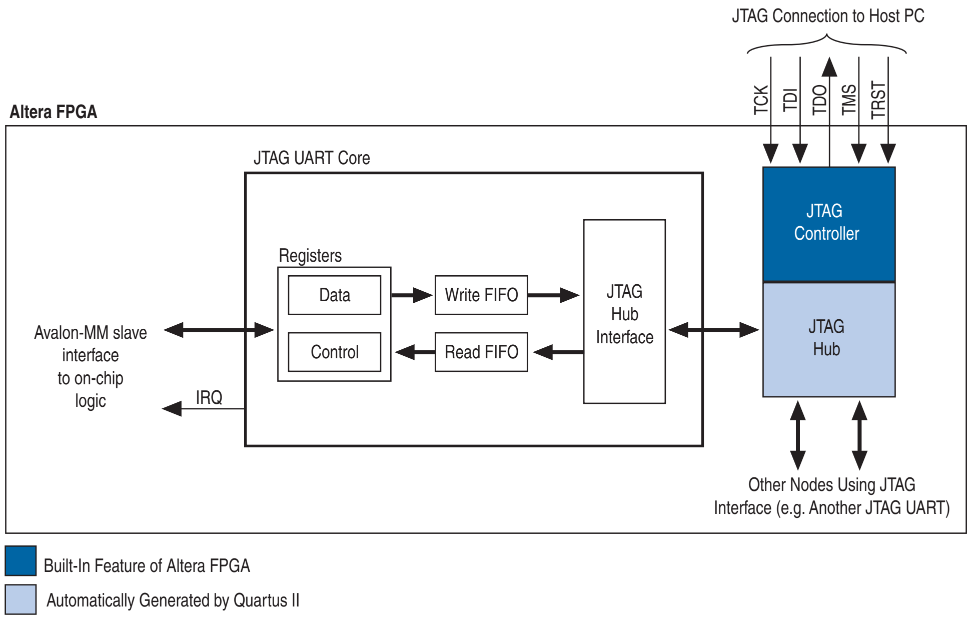
The JTAG UART core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 8–2
- "Device and Tools Support" on page 8–4
- "Instantiating the Core in SOPC Builder" on page 8–4
- "Hardware Simulation Considerations" on page 8–7
- "Software Programming Model" on page 8–7

Functional Description

Figure 8–1 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 8–1. JTAG UART Core Block Diagram


JTAG Connection to Host PC
TCK
TDI
TDO
TMS
TRST
Altera FPGA
JTAG UART Core
Registers
Data
Control
Write FIFO
Read FIFO
JTAG Hub Interface
JTAG Controller
JTAG Hub
Avalon-MM slave interface to on-chip logic
IRQ
Other Nodes Using JTAG Interface (e.g. Another JTAG UART)
Built-In Feature of Altera FPGA
Automatically Generated by Quartus II

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 8–13.

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.

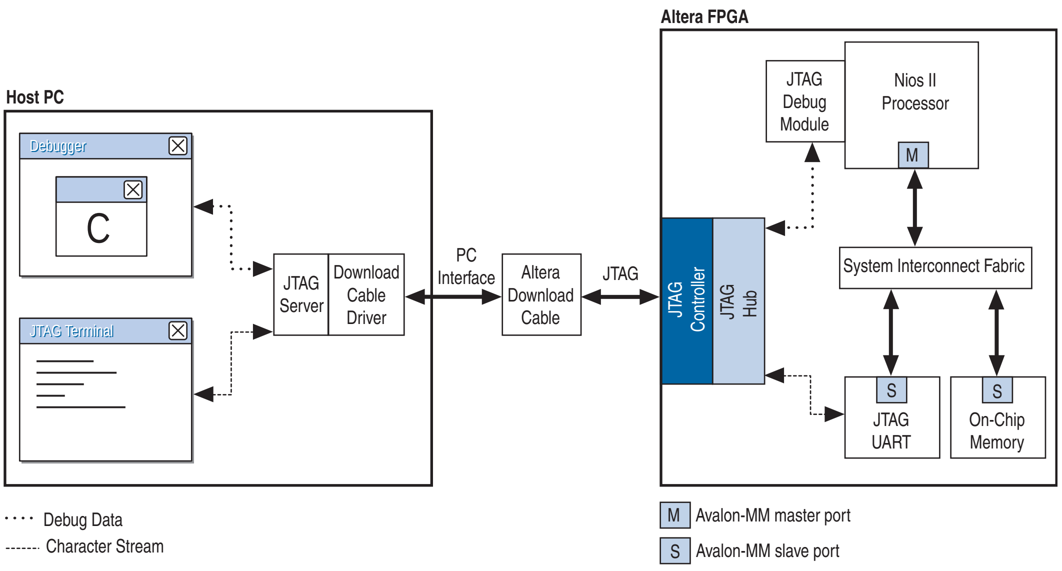
JTAG Interface

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 (that 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 8–2 shows the connection between a host PC and an SOPC Builder-generated system containing a JTAG UART core.

Figure 8–2. 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 8–2 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 the Arria™ GX, Stratix® III, Stratix II, Stratix II GX, Stratix GX, Stratix, Cyclone® III, Cyclone II, and Cyclone device families. The JTAG UART core is supported by the Nios II hardware abstraction layer (HAL) system library. No software support is provided for the first-generation Nios processor.

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

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

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 8–13 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 8–13 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.

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

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 8–1 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 8–1. Printing Characters to a JTAG UART Core as stdout

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

Example 8–2 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 8–2. 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.

The *Nios II Software Developer's Handbook* provides complete details of the HAL system library. The Nios II Embedded Design Suite (EDS) provides a number of software example designs that use the JTAG UART core.

Driver Options: Fast versus 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 8–1.

Table 8–1. 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 the 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 8–2 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 8–2. 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	*(1)*						DATA				
1	`control`	RW	WSPACE			*(1)*				AC	WI	RI	*(1)*			WE	RE

Note to Table 8–2:
(1) Reserved. Read values are undefined. Write zero.

Data Register

Embedded software accesses the read and write FIFOs via the `data` register. Table 8–3 describes the function of each bit.

Table 8–3. data Register Bits

Bit Number	Bit/Field Name	Read/Write/Clear	Description
0 .. 7	`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`, then the `DATA` field is valid, otherwise `DATA` is undefined.
16 .. 32	`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, then 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 8–4 describes the function of each bit.

Table 8–4. control Register Bits

Bit Number	Bit/Field Name	Read/Write/Clear	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.
16 .. 32	`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 8–13 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 1µs. With a threshold of 8, the interrupt response time must be less than 8µs. If the interrupt response time is too long, then performance will suffer. If it is too short, then interrupts will occur too frequently.

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

Referenced Document

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

Document Revision History

Table 8–5 shows the revision history for this chapter.

Table 8–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 8 was formerly Chapter 7.	—
May 2007 v7.1.0	● Chapter 7 was formerly chapter 5. ● Added Arria™ GX to "Device and Tools Support" on page 8–4. ● Added table of contents to Overview section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	Added Cyclone III and Stratix III support.	Version 7.0 of the Quartus II software added Cyclone III support.
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies. ● Changed old "Avalon switch fabric" term to "system interconnect fabric." ● Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface."	For version 6.1, added Stratix III support. Additionally, Altera released the Avalon Streaming interface, which necessitated some rephrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
December 2004 v1.2	Added Cyclone II support.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

9. UART Core

NII51010-7.2.0

Core Overview

The universal asynchronous receiver/transmitter core with Avalon® interface (UART core) 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 a simple register-mapped 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.

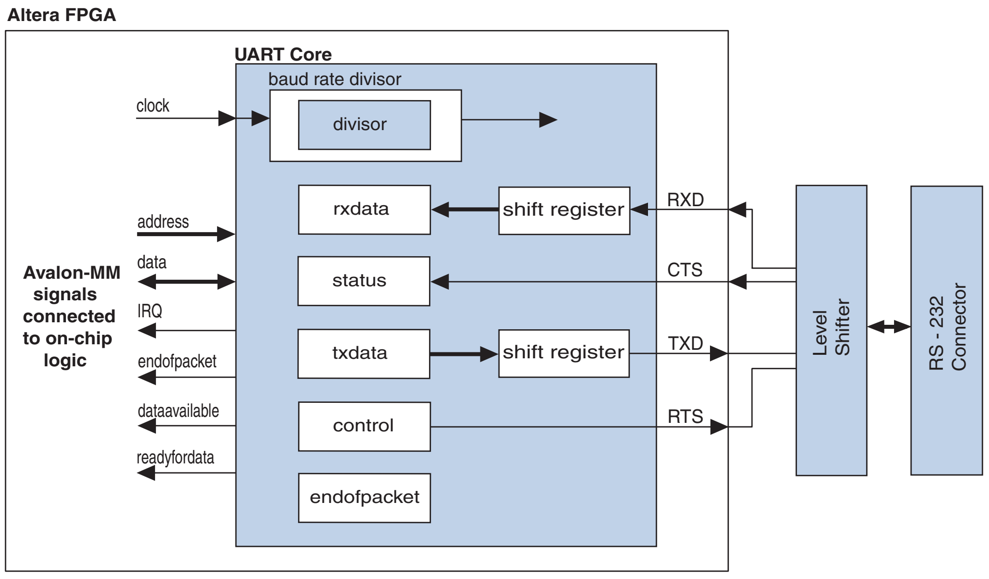
The UART core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 9–2
- "Device and Tools Support" on page 9–4
- "Instantiating the Core in SOPC Builder" on page 9–5
- "Hardware Simulation Considerations" on page 9–9
- "Software Programming Model" on page 9–9

Functional Description

Figure 9–1 shows a block diagram of the UART core.

Figure 9–1. 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 9–20.

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 the *Timer Core* chapter in volume 5 of the *Quartus II Handbook*. For details about the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification.*

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` least-significant bit (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 in the received data (frame error, parity error, receive overrun error, and break), and sets corresponding status register bits (`fe`, `pe`, `roe`, or `BRK`).

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 and Tools Support

The UART core can target all Altera FPGAs.

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.

Designers 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 re-generating the UART core hardware will result 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 (for example, 9600, 57600, 115200 bps), or you can enter any baud rate manually.

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 follows:

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

(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 (unchangeable) 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. The following settings are available.

Data Bits Setting
The settings shown in Table 9–1 are available.

Table 9–1. Data Bits Settings

Setting	Allowed 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 the Stop Bits setting.
Parity	None, Even, Odd	This setting determines whether the UART transmits characters with parity checking, and whether it expects received characters to have parity checking. Refer to "Parity Setting".

Parity Setting
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. When parity is **None**, the status register's parity error (PE) bit 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 status register's PE 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.

Flow Control

The following flow control option is available.

Include CTS/RTS Pins and Control Register Bits
When this setting is on, the UART hardware includes:

- `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 hardware listed above 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.

Avalon-MM Transfers with Flow Control (DMA)

The UART core's Avalon-MM interface optionally implements Avalon-MM transfers with flow control. This 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/from other master peripherals in the system

End-of-packet (EOP) detection allows the UART core to terminate a data transaction with a 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 resources listed above. 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 constructed. 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. The following options are available:

Create ModelSim Alias to Open Streaming Output Window
A ModelSim macro is created to open a window that displays all output from the `TXD` port.

Create ModelSim Alias to Open Interactive Stimulus Window
A ModelSim macro is created to open a window that accepts 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. This allows 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.

Hardware Simulation Considerations

The simulation features were created for easy simulation of Nios, Nios II or Excalibur™ processor systems when using the ModelSim simulator. The documentation for each 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*. For details about simulating the UART core in Nios embedded processor systems, refer to *AN 189: Simulating Nios Embedded 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 will interfere 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 9–11.

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 9–1. 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 9–2. 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 9–2 defines operation requests that the UART driver supports.

Table 9–2. UART ioctl() Operations

Request	Meaning
`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 `ioctl` "arg" parameter is ignored.
`TIOCNXCL`	Releases a previous exclusive access lock. The `ioctl` "arg" parameter is ignored.

Additional operation requests are also optionally available for the fast driver only, as shown in Table 9–3. To enable these operations in your program, you must set the preprocessor option `-DALTERA_AVALON_UART_USE_IOCTL`.

Table 9–3. Optional UART ioctl() Operations for the Fast Driver Only

Request	Meaning
`TIOCMGET`	Returns the current configuration of the device by filling in the contents of the input termios *(1)* structure. A pointer to this structure is supplied as the value of the `ioctl` "opt" parameter.
`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 `ioctl` "arg" parameter.

Note to Table 9–3:

(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**.

Refer to the *Nios II Software Developer's Handbook* for details about the `ioctl()` function.

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.

Legacy SDK Routines

The UART core is also supported by the legacy SDK routines for the first-generation Nios processor.

For details about these routines, refer to the UART documentation that accompanied the first-generation Nios processor. For details about upgrading programs based on the legacy SDK to the HAL system library API, refer to *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*.

Register Map

Programmers using the HAL API or the legacy SDK for the first-generation Nios processor 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.

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 9–4 shows the register map for the UART core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 9–4. 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	*(1)*					*(2)*	*(2)*	Receive Data						
1	`txdata`	WO	*(1)*					*(2)*	*(2)*	Transmit Data						
2	`status` *(3)*	RW	*(1)*	eop	cts	dcts	*(1)*	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	`control`	RW	*(1)*	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	`divisor` *(4)*	RW	Baud Rate Divisor													
5	`endof-packet` *(4)*	RW	*(1)*					*(2)*	*(2)*	End-of-Packet Value						

Notes to Table 9–4:
(1) These bits are reserved. Reading returns an undefined value. Write zero.
(2) 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.
(3) Writing zero to the status register clears the dcts, e, toe, roe, brk, fe, and PE bits.
(4) 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 9–5.

Table 9–5. status Register Bits (Part 1 of 3)

Bit	Bit Name	Read/ Write/ Clear	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 9–6.
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.
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.

Table 9–5. status Register Bits (Part 2 of 3)

Bit	Bit Name	Read/ Write/ Clear	Description
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 9–7.
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). Because the `CTS_N` input is logic negative, the CTS bit is 1 when a 0 logic-level is applied to the `CTS_N` input. 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 9–7.

Table 9–5. status Register Bits (Part 3 of 3)

Bit	Bit Name	Read/ Write/ Clear	Description
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 "Avalon-MM Transfers with Flow Control (DMA)" on page 9–7.

Note to Table 9–5:
(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. For example, the PE bit is bit 0 of the `status` register, and the ipe bit is bit 0 of the `control` register. An interrupt request is generated when both PE and ipe equal 1.

The control register bits are shown in Table 9–6.

Table 9–6. control Register Bits (Part 1 of 2)

Bit	Bit Name	Read/ Write	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.

Table 9–6. control Register Bits (Part 2 of 2)

Bit	Bit Name	Read/ Write	Description
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 9–7.
12	IEOP	RW	Enable interrupt for end-of-packet condition.

Note to Table 9–6:
(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 shown in Equation 3:

$$\text{baud rate} = \frac{\text{clock frequency}}{(\text{divisor} + 1)} \tag{3}$$

The `divisor` register is an optional hardware feature. If the **Baud Rate Can Be Changed By Software** hardware option is not enabled, then 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 9–5.

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 9–5 on page 9–16 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, then 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:

- *Nios II Software Developer's Handbook*
- *Timer Core chapter in volume 5 of the Quartus II Handbook*
- *Avalon Memory-Mapped Interface Specification*
- *AN 351: Simulating Nios II Processor Designs*
- *AN 189: Simulating Nios Embedded Processor Designs*

Document Revision History

Table 9–7 shows the revision history for this chapter.

Table 9–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.	● Chapter 9 was formerly Chapter 8. ● Added two sentences to clarify use of flow control. Host PC must also be configured for flow control.	—
May 2007 v7.1.0	● Chapter 8 was formerly chapter 6. ● Added table of contents to Overview section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.” ● Corrected definition of even and odd parity in section “*Data Bits, Stop Bits, Parity*” on page 8–6.	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology. Other changes to the document serve only to clarify existing behavior.
May 2006 v6.0.0	No change from previous release.	—
December 2005 v5.1.1	Changed Avalon “streaming” terminology to “flow control” based on a change to the *Avalon Interface Specification.*	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

ALTERA®

10. SPI Core

NII51011-7.2.0

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 16 independent SPI slaves. The width of the receive and transmit registers are configurable between 1 and 16 bits. Longer transfer lengths (for example, 24-bit transfers) 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 10–7
- "Device and Tools Support" on page 10–10
- "Software Programming Model" on page 10–10

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

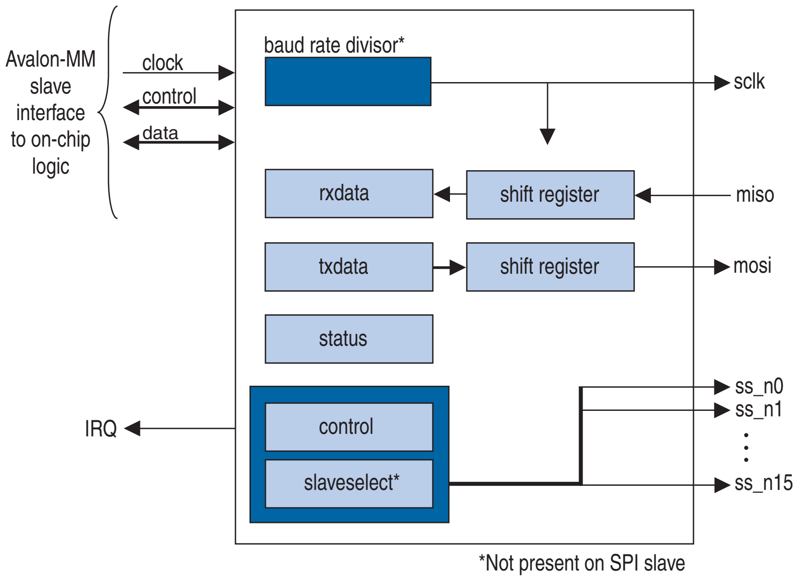
The SPI core has the following user-visible features:

- 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 10–1 shows a block diagram of the SPI core in master mode.

Figure 10–1. SPI Core Block Diagram

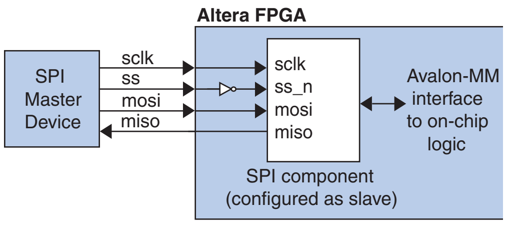


The SPI core logic is synchronous to the clock input provided by the Avalon-MM interface. When configured as a master, the core divides the Avalon-MM clock to generate the SCLK output. When configured as a slave, the core's receive logic is synchronized to SCLK input. The core's Avalon-MM interface is 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. See the *Timer Core* chapter for details.

Example Configurations

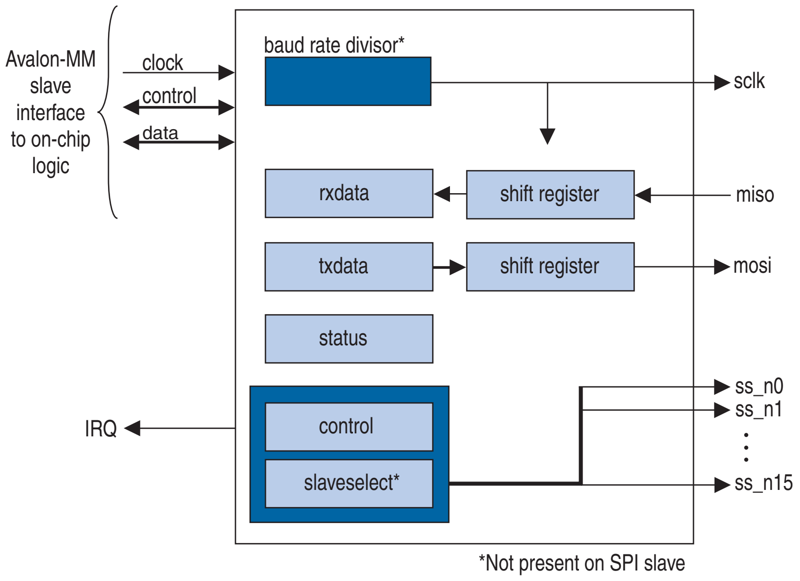
Two possible configurations are shown below. In Figure 10–2, the SPI core provides a slave interface to an off-chip SPI master.

Figure 10–2. SPI Core Configured as a Slave


Altera FPGA
SPI Master Device
sclk
ss
mosi
miso
sclk
ss_n
mosi
miso
Avalon-MM interface to on-chip logic
SPI component (configured as slave)

In Figure 10–3 the SPI core provides a master interface driving multiple off-chip slave devices. Each slave device in Figure 10–3 must tristate its `miso` output whenever its select signal is not asserted.

Figure 10–3. SPI Core Configured as a Master



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 1 to 16. 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 least-significant bit (LSB) first or most-significant bit (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 1 to 16. 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 during data receiving. 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 least-significant bit (LSB) first or most-significant bit (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 10–1.

Table 10–1. Master Mode Port Configurations

Name	Direction	Description
`mosi`	output	Data output to slave(s)
`miso`	input	Data input from slave(s)
`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 15.

Only an SPI master can initiate an operation between master and slave. 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 (that is, 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, up to a maximum of sixteen slaves. 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 conflict 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 10–2.

Table 10–2. 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

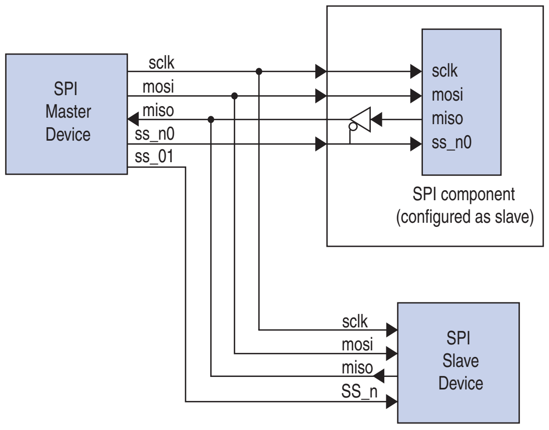
In slave mode, the SPI core simply waits for the master to initiate transactions. Before a transaction begins, the slave logic is continuously polling the `ss_n` input. When the master asserts `ss_n` (drives it low), the slave logic immediately begins sending the transmit shift register contents to the `miso` output. The slave logic also captures data on the `mosi` input, and fills the receive shift register simultaneously. Thus, a read and write transaction are carried out simultaneously.

An intelligent host (for example, a microprocessor) writes data to the `txdata` registers, so that it will be 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 will be 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 10–4 shows an example of the SPI core in slave mode in an environment with two slaves.

Figure 10–4. 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.

Instantiating the SPI Core in SOPC Builder

Designers 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:

- "Generate Select Signals"
- "SPI Clock (sclk) Rate" on page 10–8
- "Specify Delay" on page 10–8

Generate Select Signals

This setting specifies how many slaves the SPI master will connect to. The acceptable range is 1 to 16. 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:

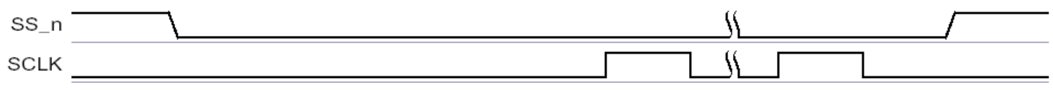
<Avalon-MM system clock frequency> / [2, 4, 6, 8, ...]

The actual frequency achieved will not be greater than the specified target value. For example, if the system clock frequency is 50 MHz and the target value is 25 MHz, then the clock divisor is 2 and the actual `sclk` frequency achieves exactly 25 MHz. However, if the target frequency is 24 MHz, then the clock divisor is 4 and the actual `sclk` frequency becomes 12.5 MHz.

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, the designer must also specify the delay time in units of ns, us or ms. An example is shown in Figure 10–5.

Figure 10–5. 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 the following equations:

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

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

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. Acceptable values are from 1 to 16.
- *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.

Figures 10–6 through 10–9 demonstrate the behavior of signals in all possible cases of clock polarity and clock phase.

Figure 10–6. Clock Polarity = 0, Clock Phase = 0

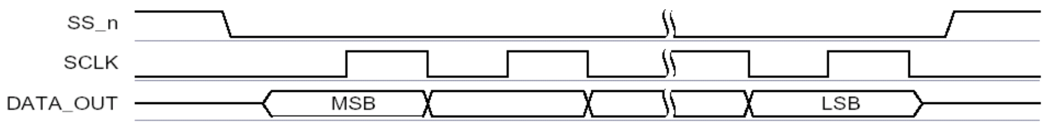


Figure 10–7. Clock Polarity = 0, Clock Phase = 1

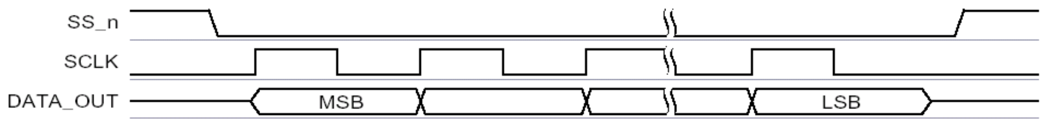


Figure 10–8. Clock Polarity = 1, Clock Phase = 0

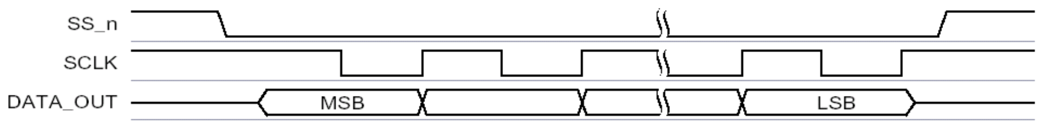
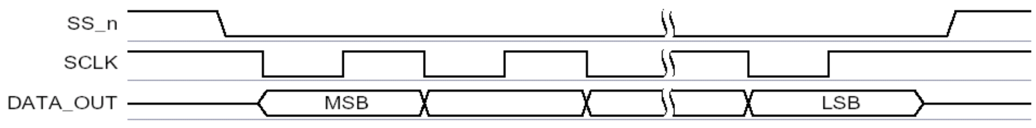


Figure 10–9. Clock Polarity = 1, Clock Phase = 1



Device and Tools Support

The SPI core can target all Altera FPGAs.

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 an SPI core 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: `alt_avalon_spi_command()` is used to perform a control sequence on the SPI bus. This routine is designed for SPI masters of 8-bit data width or less. Currently, it does not support SPI hardware with data-width greater than 8 bits. A single call to this function writes a data buffer of arbitrary length out 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 numbered 0, the next is 1, etc.
(2) Transmits `write_length` bytes of data from `wdata` through the SPI interface, discarding the incoming data on MISO.
(3) Reads `read_length` bytes of data, storing the data into the buffer pointed to by `read_data`. MOSI 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 then you can call the function multiple times, specifying the merge flag on all the accesses except the last.

This function is not thread safe. If you want to access the SPI bus 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.

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.

Legacy SDK Routines

The SPI core is also supported by the legacy SDK routines for the first-generation Nios processor. For details about these routines, refer to the SPI documentation that accompanied the first-generation Nios processor.

☞ For details about upgrading programs based on the legacy SDK to the HAL system library API, refer to *AN 350: Upgrading Nios Processor Systems to the Nios II Processor.*

Register Map

An Avalon-MM master peripheral controls and communicates with the SPI core via the six 16-bit registers, shown in Table 10–3. The table assumes an *n*-bit data width for `rxdata` and `txdata`.

Table 10–3. Register Map for SPI Master Device

Internal Address	Register Name	15...11	10	9	8	7	6	5	4	3	2	1	0
0	`rxdata` *(1)*	RXDATA (n-1..0)											
1	`txdata` *(1)*	TXDATA (n-1..0)											
2	`status` *(2)*				E	RRDY	TRDY	TMT	TOE	ROE			
3	`control`		sso *(3)*		IE	IRRDY	ITRDY		ITOE	IROE			
4	Reserved												
5	`slaveselect` *(3)*	Slave Select Mask											

Notes to Table 10–3:
(1) Bits 15 to *n* are undefined when *n* is less than 16.
(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 10–14.

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 10–4 describes the individual bits of the `status` register.

Table 10–4. 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** The `TMT` bit is set to 0 when a transaction is in progress and set to 1 when the shift register is empty.
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 10–5.

Table 10–5. 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 (in other words, greater than 16 bits).

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

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. For example, to enable communication with slave device 3, set bit 3 of `slaveselect` to 1.

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 Document

This chapter references AN 350: Upgrading Nios Processor Systems to the Nios II Processor.

Document Revision History

Table 10–6 shows the revision history for this chapter.

Table 10–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	● Chapter 10 was formerly chapter 9. ● Added "Referenced Document" on page 10–15.	—
May 2007 v7.1.0	● Chapter 9 was formerly chapter 7. ● Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies ● Changed old "Avalon switch fabric" term to "system interconnect fabric" ● Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface"	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
December 2005 v5.1.1	Changed Avalon “"streaming” terminology to “flow control” based on a change to the *Avalon Interface Specification.*	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

Section III. Display Peripherals

This section describes display interface peripherals provided by Altera®. These components provide interfaces to visual display devices for SOPC Builder systems.

See *About This Handbook* for further details.

This section includes the following chapters:

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

11. Optrex 16207 LCD Controller Core

NII51019-7.2.0

Core Overview

The Optrex 16207 LCD controller core with Avalon® Interface (“the LCD controller”) 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 11–2
- “Instantiating the Core in SOPC Builder” on page 11–2
- “Software Programming Model” on page 11–2

Functional Description

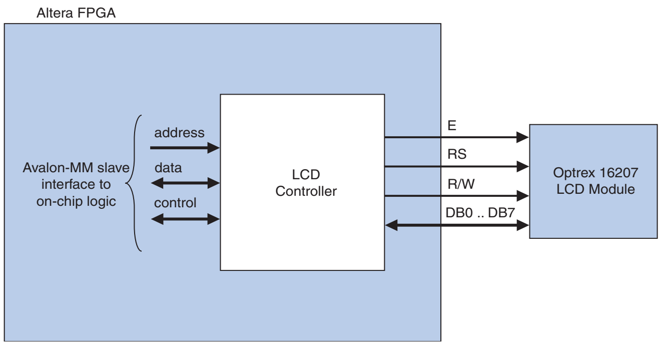
The LCD controller hardware consists of two user-visible components:

- 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 — The HAL device drivers make it unnecessary for users to access the registers directly. Therefore, Altera does not provide details about the register usage. For further details, refer to “Software Programming Model” on page 11–2.

Figure 11–1 shows a block diagram of the LCD controller core.

Figure 11–1. LCD Controller Block Diagram



Device and Tools Support

The LCD controller hardware supports all Altera FPGA families. The LCD controller drivers support the Nios II processor. The drivers do not support the first-generation Nios processor.

Instantiating the Core in SOPC Builder

In SOPC Builder, the LCD controller component has the name Character LCD (16×2, Optrex 16207). The LCD controller does not have any user-configurable settings. The only choice to make in SOPC Builder is whether or not to add an LCD controller to the system. For each LCD controller included in the system, the top-level system module includes the 11 signals that connect to the LCD module.

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 will fit on the display, then all characters are displayed. If the buffer is wider than the display, then 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 understands 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 11–1.

Table 11–1. 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 will return 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, then 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 Document

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

Document Revision History

Table 11–2 shows the revision history for this chapter.

Table 11–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 11 was formerly chapter 10.	—
May 2007 v7.1.0	● Chapter 10 was formerly chapter 8. ● Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies ● Changed old “Avalon switch fabric” term to “system interconnect fabric” ● Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	Chapter title changed, but no change in content from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.0	Initial release.	—

12. Video Sync Generator and Pixel Converter Cores

QII55006-7.2.0

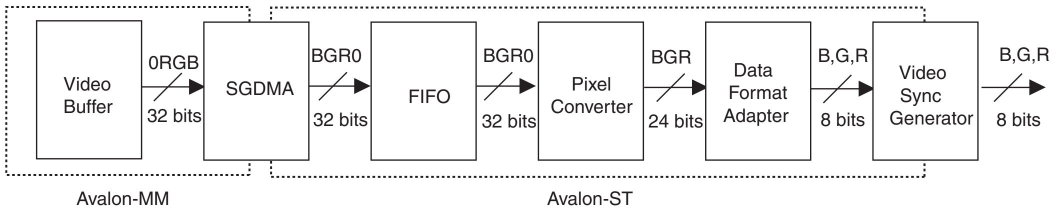
Core Overview

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 12–1 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 12–1. 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 (EEK), which includes an LCD display daughtercard assembly attached via an HSMC connector.

This chapter contains the following sections:

- "Video Sync Generator" on page 12–2
- "Pixel Converter" on page 12–5
- "Device and Tools Support" on page 12–6
- "Hardware Simulation Considerations" on page 12–7

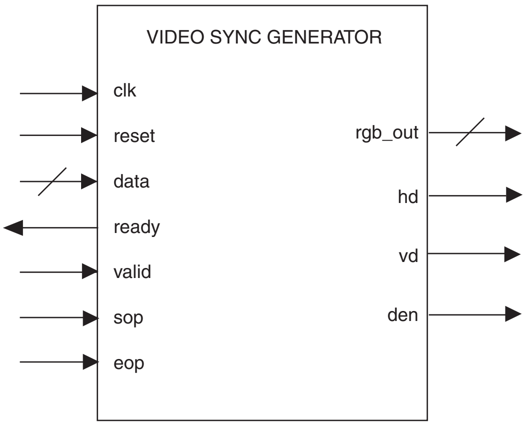
Video Sync Generator

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-ST input interface and outputs the data to an off-chip display controller. No processing or validation is performed on the pixel data. Figure 12–2 shows a block diagram of the video sync generator.

Figure 12–2. 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 12–3 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 12–1 lists the available parameters in the MegaWizard interface.

Table 12–1. Video Sync Generator Parameters

Parameter Name	Description
Data Stream Bit Width	The width of the inbound and outbound data.
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.
Number of Columns	The number of active pixels in each line.
Number of Rows	The number of active scan lines in each video frame.
Horizontal Blank Pixels	The number of blanking pixels that preceed the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
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 Blank Lines	The number of blanking lines that preceed the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Vertical Front Porch Pixels	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.
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**.
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**.

Signals

Table 12–2 lists the input and output signals for the video sync generator core.

Table 12–2. Video Sync Generator Core Signals (Part 1 of 2)

Signal Name	Width (Bits)	Direction	Description
Global Signals			
`clk`	1	Input	System clock.
`reset`	1	Input	System reset.
Avalon-ST Signals			

Table 12–2. Video Sync Generator Core Signals (Part 2 of 2)

Signal Name	Width (Bits)	Direction	Description
`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 12–3 shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 8 and 3, respectively.

Figure 12–3. Horizontal Synchronization Timing—8 bits DataWidth and 3 Beats Per Pixel

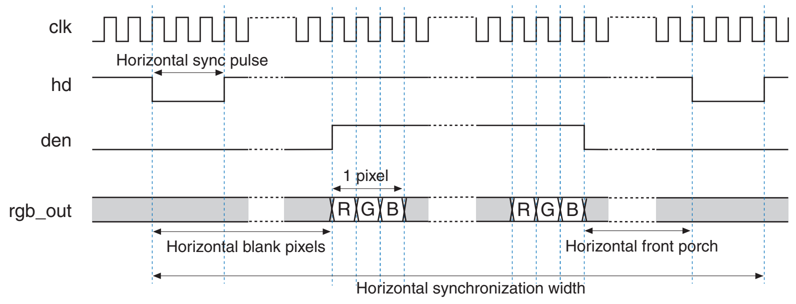


Figure 12–4 shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 24 and 1, respectively.

Figure 12–4. Horizontal Synchronization Timing—24 bits Datawidth and 1 Beat Per Pixel

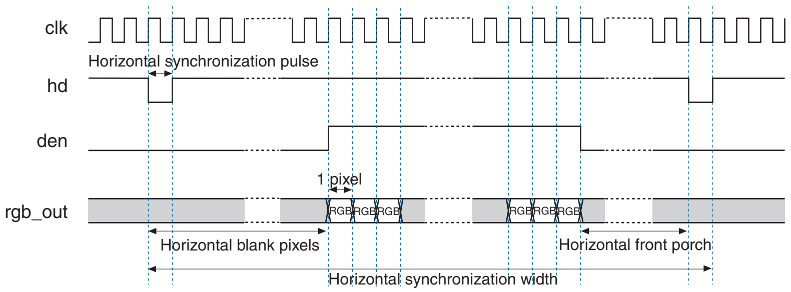
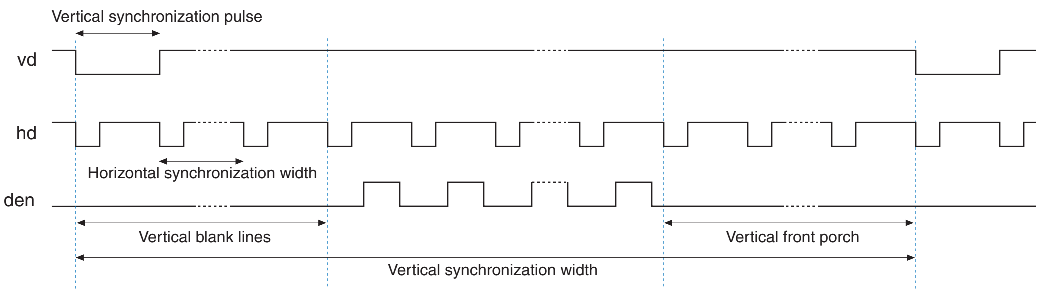


Figure 12–5 shows the vertical synchronization timing.

Figure 12–5. Vertical Synchronization Timing


Vertical synchronization pulse
vd
hd
Horizontal synchronization width
den
Vertical blank lines
Vertical front porch
Vertical synchronization width

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. There are no user-configurable settings for this core.

Signals

Table 12–3 lists the input and output signals for the pixel converter core.

Table 12–3. 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 tranferred 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 and Tools 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 ms to get a full video frame. Depending on the size of the video frame, simulation may take a very long time to complete.

Referenced Document

This chapter references the *Avalon Streaming Interface Specification*.

Document Revision History

Table 12–4 shows the revision history for this chapter.

Table 12–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Initial release.	—

Section IV. Multiprocessor Coordination 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.

See *About This Handbook* for further details.

This section includes the following chapters:

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

13. Mutex Core

NII51020-7.2.0

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 and Tools Support" on page 13–2
- "Instantiating the Core in SOPC Builder" on page 13–2
- "Software Programming Model" on page 13–2
- "Mutex API" on page 13–4

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 13–1 shows the registers.

Table 13–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 … 16	15 … 1	0
0	`mutex`	RW	OWNER	VALUE	
1	`reset`	RW	–	–	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 available (i.e, unlocked). Otherwise, the mutex is unavailable (i.e., locked).
- The `mutex` register is always readable. A processor (or any Avalon-MM master peripheral) 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 is 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 VALUE. 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 and Tools Support

The mutex 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

Hardware designers 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, such as the software constructs used to access the hardware. 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**—this file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mutex.h—**this file defines data structures and functions to access the mutex core hardware.
- **altera_avalon_mutex.c**—this file contains the implementations of the functions to access the mutex core

Hardware Mutex

This section describes the low-level software constructs for manipulating the mutex core hardware.

The file **altera_avalon_mutex.h** declares a structure `alt_mutex_dev` that represents an instance of a mutex device. It also declares functions for accessing the mutex hardware structure, listed in Table 13–2.

Table 13–2. Hardware Mutex Functions

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 13–4.

Example 13–1 demonstrates opening a mutex device handle and locking a mutex:

Example 13–1. 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:	Zero if the mutex was successfully locked, or non zero if 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:	-
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 13–3 shows the revision history for this chapter.

Table 13–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 13 was formerly chapter 11.	—
May 2007 v7.1.0	● Chapter 11 was formerly chapter 9. ● Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies ● Changed old “Avalon switch fabric” term to “system interconnect fabric” ● Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
December 2004 v1.0	Initial release.	—

14. Mailbox Core

NII53001-7.2.0

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 and Tools Support" on page 14–2
- "Instantiating the Core in SOPC Builder" on page 14–2
- "Software Programming Model" on page 14–3
- "Mailbox API" on page 14–6

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 14–1 shows the registers.

Table 14–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 … 16	15 … 1	0
0	`mutex0`	RW	OWNER	VALUE	
1	`reset0`	RW	–	–	RESET
2	`mutex1`	RW	OWNER	VALUE	
3	`reset1`	RW	–	–	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 14–3 for details about how to use the mailbox core in software.

For a detailed description of the mutex hardware operation, refer to the Mutex Core chapter in volume 5 of the *Quartus II Handbook*.

Device and Tools Support

The mailbox 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

Hardware designers instantiate and configure the mailbox core in an SOPC Builder system using the following process:

1. Decide which processors will 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 14–3.
   - **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 14–3.
   - **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, such as the software constructs used to access the hardware. 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 14–2. For a complete description of each function, refer to "Mailbox API" on page 14–6.

Table 14–2. 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 14–1 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 14–1. Example: 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:	–
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.

Referenced Document

This chapter references the *Mutex Core* chapter in volume 5 of the *Quartus II Handbook*

Document Revision History

Table 14–3 shows the revision history for this chapter.

Table 14–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 14 was formerly chapter 12.	—
May 2007 v7.1.0	● Chapter 12 was formerly chapter 10. ● Revised "Instantiating the Core in SOPC Builder" on page 14–2 to reflect the GUI changing from the More Settings tab to the MegaWizard interface. ● Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies. ● Changed old "Avalon switch fabric" term to "system interconnect fabric." ● Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface."	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	Initial release.	—

Section V. Other Memory-Mapped Peripherals

This section describes other peripherals provided by Altera® for SOPC Builder systems.

See *About This Handbook* for further details.

This section includes the following chapters:

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

15. PIO Core

NII51007-7.2.0

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)

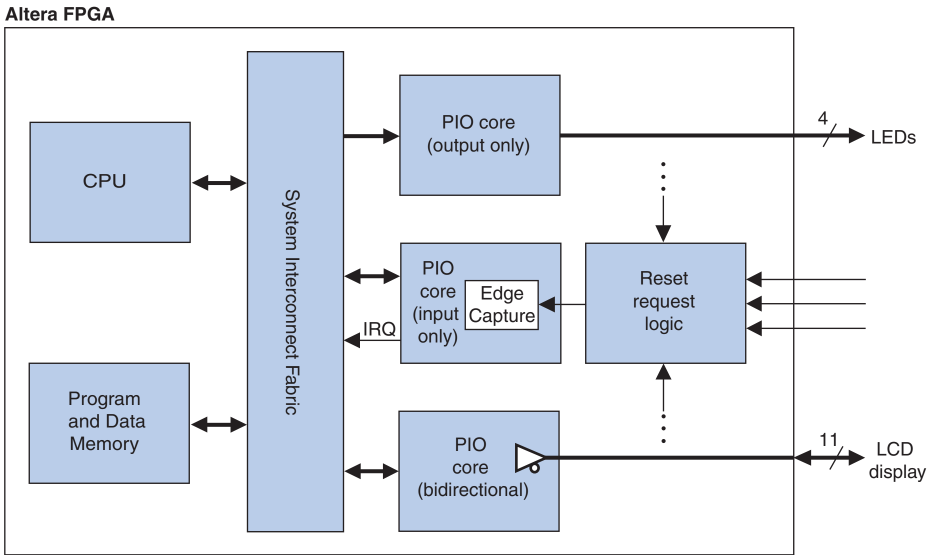
The PIO core interrupt request (IRQ) output can assert an interrupt based on input signals. The PIO core is SOPC Builder ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- Functional Description
- "Example Configurations" on page 15–4
- "Instantiating the PIO Core in SOPC Builder" on page 15–5
- "Device and Tools Support" on page 15–6
- "Software Programming Model" on page 15–6

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 15–1 shows an example of a processor-based system that uses multiple PIO cores to blink LEDs, capture edges from on-chip reset-request control logic, and control an off-chip LCD display.

Figure 15–1. 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 15–2 on page 15–7 for a description of the registers. Some registers are not necessary in certain hardware configurations, in which case the unnecessary registers do not exist. Reading a non-existent register returns an undefined value, and writing a non-existent register has no effect.

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 will be 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 type of edges to detect 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 15–2 shows a block diagram of the PIO core configured with input and output ports, as well as support for IRQs.

Figure 15–2. PIO Core with Input Ports, Output Ports and IRQ Support

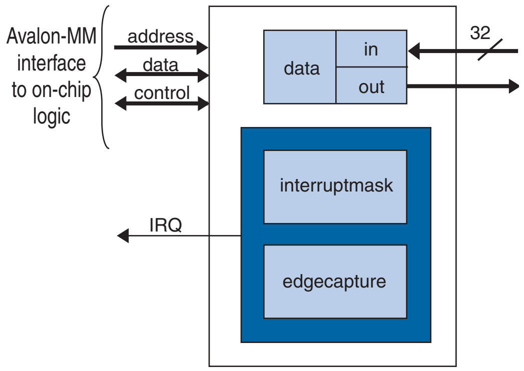
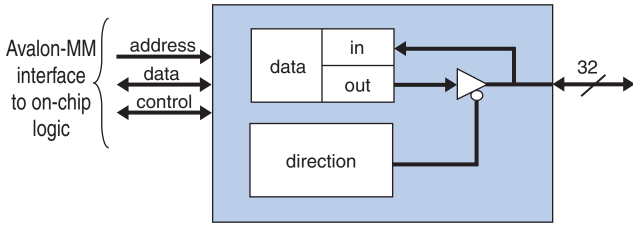


Figure 15–3 shows a block diagram of the PIO core configured in bidirectional mode, without support for IRQs.

Figure 15–3. 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

Designers use the MegaWizard® interface for the PIO core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.

The MegaWizard interface has two tabs, **Basic Settings** and **Input Options**.

Basic Settings

The **Basic Settings** page allows the designer to specify the width and direction of the I/O ports.

- The **Width** setting can be any integer value between 1 and 32. For a value of *n*, the I/O ports become *n*-bits wide.
- The **Direction** setting has four options, as shown in Table 15–1.

Table 15–1. 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.

Input Options

The **Input Options** page allows the designer 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

Synchronously Capture
When **Synchronously capture** is on, the PIO core contains the edge capture register, `edgecapture`. The user must further specify what type of edge(s) to detect:

- **Rising Edge**
- **Falling Edge**
- **Either Edge**

The edge capture register allows the core to detect and (optionally) generate an interrupt when an edge of the specified type occurs on an input port.

When **Synchronously capture** is off, the `edgecapture` register does not exist.

Enable Bit Clearing for Edge Capture Register
Turning on **Enable bit-clearing for edge capture register** allows you to clear individual bit(s) in the edge capture register. To clear a given bit, write 1 to the bit in the edge capture register. Example—To clear bit 6 in the edge capture register, write 01000000 to the register.

Interrupt

When **Generate IRQ** is on, the PIO core is able 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.

Device and Tools Support

The PIO core supports all Altera® FPGA 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.

Legacy SDK Routines

The PIO core is supported by the legacy SDK routines for the first-generation Nios processor. For details about these routines, refer to the PIO documentation that accompanied the first-generation Nios processor.

For details about upgrading programs based on the legacy SDK to the HAL system library API, refer to *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*.

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 15–2. The table assumes that the PIO core's I/O ports are configured to a width of *n* bits.

Table 15–2. 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.				

Notes to Table 15–2:

(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 re-program 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 15–9.

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. Writing any value to `edgecapture` clears all bits in the register.

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.

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.

Referenced Document

This chapter references AN 350: Upgrading Nios Processor Systems to the Nios II Processor.

Document Revision History

Table 15–3 shows the revision history for this chapter.

Table 15–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	● Chapter 15 was formerly chapter 13. ● Added the description for a new parameter, **Enable Bit Clearing for Edge Capture Register**.	—
May 2007 v7.1.0	● Chapter 13 was formerly chapter 11. ● Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies ● Changed old “Avalon switch fabric” term to “system interconnect fabric” ● Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

16. Timer Core

NII51008-7.2.0

Core Overview

The timer core with Avalon® interface is a 32-bit interval timer for Avalon-based processor systems, such as a Nios® II processor system. The timer provides the following features:

- Controls to start, stop, and reset the timer
- Two count modes: count down once and continuous count-down
- Count-down period register
- Maskable interrupt request (IRQ) upon reaching 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

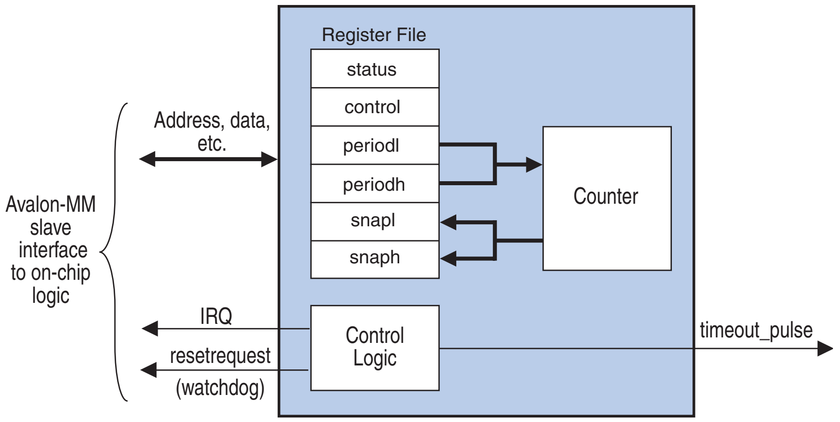
Device drivers are provided in the HAL system library for the Nios II processor. The timer core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 16–2
- "Device and Tools Support" on page 16–3
- "Instantiating the Core in SOPC Builder" on page 16–3
- "Software Programming Model" on page 16–6

Functional Description

Figure 16–1 shows a block diagram of the timer core.

Figure 16–1. Timer Core Block Diagram



The 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 timer compatible with both 16-bit and 32-bit processors. Certain registers only exist in hardware for a given configuration. For example, if the timer is configured with a fixed period, the period registers do not exist in hardware.

The basic behavior of the timer is described below:

- An Avalon-MM master peripheral, such as a Nios II processor, writes the timer core's `control` register to:
  - 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, `periodl` and `periodh`.
- 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 either `snapl` or `snaph` to request a coherent snapshot of the counter, and then reading `snapl` and `snaph` for the full 32-bit value.
- When the count reaches zero:
  - 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 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 16–5.

Device and Tools Support

The timer core supports all Altera® FPGA families.

Instantiating the Core in SOPC Builder

Designers use the MegaWizard® interface for the timer core in SOPC Builder to specify the hardware features. This section describes the options available in the MegaWizard interface.

Timeout Period

The **Timeout Period** setting determines the initial value of the `periodl` and `periodh` registers. When the **Writeable period** setting is enabled, a processor can change the value of the period by writing `periodl` and `periodh`. When **Writeable period** (see below) is off, the period is fixed and cannot be updated at runtime.

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**, then the true timeout period will be 1.020 microseconds.

Hardware Options

The following options affect the hardware structure of the 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 16–5.

Register Options

Table 16–1 shows the settings that affect the timer core's registers.

Table 16–1. Register Options

Option	Description
Writeable period	When this option is enabled, a master peripheral can change the count-down period by writing `periodl` and `periodh`. When disabled, the count-down period is fixed at the specified **Timeout Period**, and the `periodl` and `periodh` 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 `snapl` and `snaph` 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 16–2 shows the settings that affect the timer core's output signals.

Table 16–2. Output Signal Options

Option	Description
Timeout pulse (1 clock wide)	When this option is enabled, the timer core outputs a signal `timeout_pulse`. This signal pulses high for one clock cycle whenever the timer reaches zero. When disabled, the `timeout_pulse` signal does not exist.
System reset on timeout (watchdog)	When this option is enabled, the timer core's Avalon-MM slave port includes the `resetrequest` signal. This signal pulses high for one clock cycle (causing a system-wide reset) whenever the timer reaches zero. When this option is enabled, the internal timer is stopped at reset. Explicitly writing the START bit of the `control` register starts the timer. When this option is disabled, 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 timer 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 (i.e., 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 either the `periodl` or `periodh` registers (the written value is ignored). If the processor fails to access the timer because, for example, software stopped executing normally, then 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 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 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 timer via the HAL API, rather than accessing the timer registers.

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 timer 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 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, then 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 timer core.

Limitations

The HAL driver for the timer core does not support the watchdog reset feature of the timer core.

Software Files

The 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

A programmer should never have to directly access the timer 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 16–3 shows the register map for the timer.

Table 16–3. Register Map (Part 1 of 2)

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)						

Table 16–3. Register Map (Part 2 of 2)

Offset	Name	R/W	Description of Bits						
			15	…	4	3	2	1	0
4	`snapl`	RW	Counter Snapshot (bits 15..0)						
5	`snaph`	RW	Counter Snapshot (31..16)						

Note to Table 16–3:
(1) Reserved. Read values are undefined. Write zero.

status Register

The `status` register has two defined bits, as shown in Table 16–4.

Table 16–4. status Register Bits

Bit	Name	Read/ Write/ Clear	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 16–5.

Table 16–5. control Register Bits (Part 1 of 2)

Bit	Name	Read/ Write/ Clear	Description
0	ITO	RW	If the ITO bit is 1, the 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 32-bit value stored in the `periodl` and `periodh` registers, regardless of the CONT bit.

Table 16–5. control Register Bits (Part 2 of 2)

Bit	Name	Read/ Write/ Clear	Description
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 held 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.
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. Writing 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 16–5:
(1) Writing 1 to both START and STOP bits simultaneously produces an undefined result.

periodl and periodh Registers

The `periodl` and `periodh` registers together store the timeout period value. `periodl` holds the least-significant 16 bits, and `periodh` holds the most-significant 16 bits. The internal counter is loaded with the 32-bit value stored in `periodh` and `periodl` whenever one of the following occurs:

- A write operation to either the `periodh` or `periodl` register
- The internal counter reaches 0

The timer's actual period is one cycle greater than the value stored in `periodh` and `periodl`, because the counter assumes the value zero (0×00000000) for one clock cycle.

Writing to either `periodh` or `periodl` 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 either `periodh` or `periodl` causes the counter to reset to the fixed **Timeout Period** specified at system generation time.

snapl and snaph Registers

A master peripheral may request a coherent snapshot of the current 32-bit internal counter by performing a write operation (write-data ignored) to either the `snapl` or `snaph` registers. When a write occurs, the value of the counter is copied to `snapl` and `snaph`. `snapl` holds the least-significant 16 bits of the snapshot and `snaph` holds the most-significant 16 bits. The snapshot occurs whether or not the counter is running. Requesting a snapshot does not change the internal counter's operation.

Interrupt Behavior

The 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 Document

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

Document Revision History

Table 16–6 shows the revision history for this chapter.

Table 16–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	● Chapter 16 was formerly chapter 14. ● Updated and expanded definition of Timeout Period	—
May 2007 v7.1.0	● Corrected an error: The timer can be used as a timestamp device if it has a writeable *period* register. ● Added table of contents to Overview section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon switch fabric” term to “system interconnect fabric.” Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.” ● Added statement that failure to acknowledge an IRQ results in an undefined result in section “Interrupt Behavior” on page 12–9.	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

NII51014-7.2.0

17. 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

The system ID core provides a read-only Avalon Memory-Mapped (Avalon-MM) slave interface. This interface has two registers, as shown in Table 17–1.

Table 17–1. System ID Core Register Map

Offset	Register Name	R/W	Bit Description
			31…0
0	`id`	R	SOPC Builder System ID *(1)*
1	`timestamp`	R	SOPC Builder Generation Time *(1)*

Note to Table 17–1:
(1) Return value is constant.

The value of each register is determined at system generation time, and always returns a constant value. The meaning of the values is:

- `id`— 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.

- `timestamp`—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, then 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 and Tools Support

The system ID core supports all device families supported by SOPC Builder. The system ID core provides a device driver for the Nios II hardware abstraction layer (HAL) system library. No software support is provided for any other processor, including the first-generation Nios processor.

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® Plug-In Manager interface for the System ID core. Hovering the mouse over the component in SOPC Builder also displays a tool-tip showing the values.

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 17–2 shows the revision history for this chapter.

Table 17–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 17 was formerly chapter 15.	—
May 2007 v7.1.0	● Chapter 15 was formerly chapter 13. ● Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies ● Changed old “Avalon switch fabric” term to “system interconnect fabric” ● Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

18. PLL Core

NII53002-7.2.0

Core Overview

The Avalon® memory-mapped (Avalon-MM) phase locked loop (PLL) core with Avalon interface provides a means of accessing the dedicated on-chip PLL circuitry in Altera's Stratix® and Cyclone® series FPGAs. The PLL core is a component wrapper around the Altera® altpll Megafunction.

The core takes an SOPC Builder system clock as its input and generates PLL output clocks locked to that reference clock.

The PLL core supports 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-MM registers or top-level signals on the SOPC Builder system module.

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

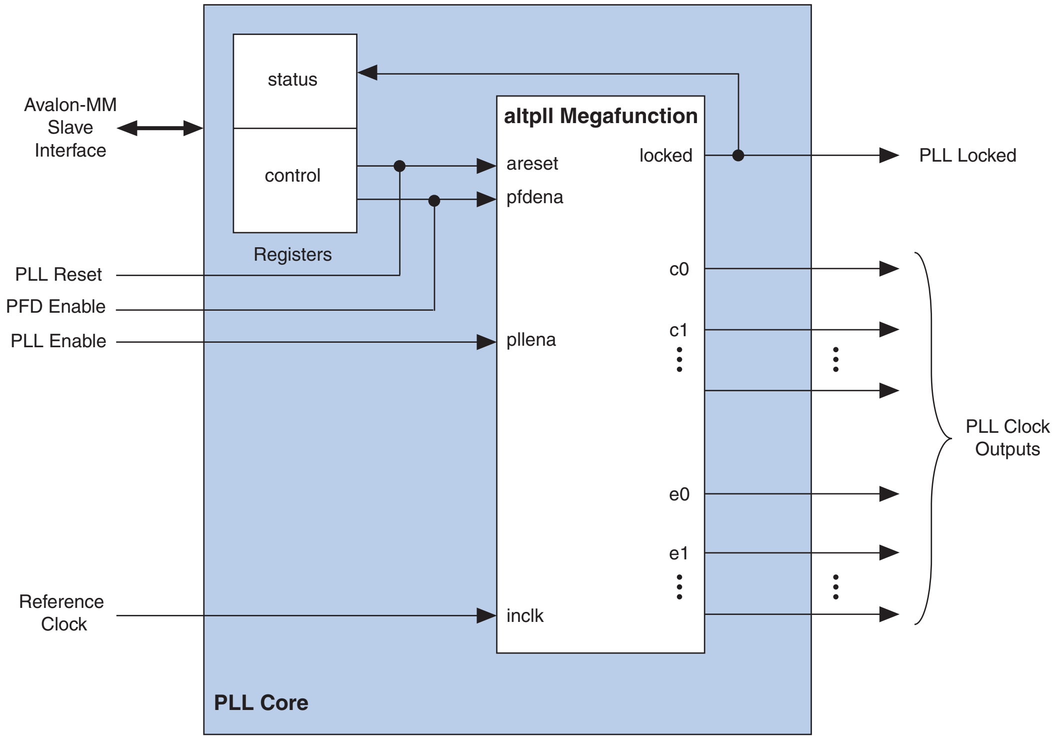
For details about the altpll megafunction, refer to the *altpll Megafunction User Guide*.

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

Functional Description

Figure 18–1 shows a block diagram of the PLL core and its connection to the PLL circuitry inside an Altera FPGA. The following sections describe the components of the core.

Figure 18–1. PLL Core Block Diagram



altpll Megafunction

The PLL core consists of an altpll megafunction instantiation and an Avalon-MM slave interface. This interface can optionally provide access to status and control registers within the core. The altpll Megafunction takes an SOPC Builder system clock as its reference, and generates one or more phase-locked 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 can not be used as on-chip clock sources. External clock outputs are not available on all device families.

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 Core in SOPC Builder" on page 18–4.

System Reset Considerations

At FPGA configuration, the PLL core resets 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 and Tools Support

The PLL core is supported by the Quartus II software version 5.1 and later. The core supports any Altera FPGA family supported by the altpll megafunction.

For more information about the altpll megafunction, refer to the *altpll Megafunction User Guide*.

Instantiating the Core in SOPC Builder

The PLL core contains an instantiation of the altpll Megafunction. The MegaWizard® interface for the PLL core allows you to configure the altpll, and specify connections to selected altpll status and control signals. The PLL core appears in the **Other** category in the SOPC Builder list of available components.

The following sections describe the options available in the MegaWizard interface for the Avalon-MM PLL core in SOPC Builder.

PLL Settings Page

The **PLL Settings** page contains a button that launches Altera's altpll MegaWizard Plug-In Manager. Use the MegaWizard interface to parameterize the altpll megafunction. The set of available parameters depends on the target device family.

For details about using the altpll MegaWizard interface, refer to the *altpll Megafunction User Guide*.

You cannot click **Finish** in the Avalon-MM 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, 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, the PLL's default behavior will be as shown in Table 18–1.

You can specify the access mode for the advanced signals shown in Table 18–1. The altpll core signals, not displayed in this table, are automatically exported to the top level of the SOPC Builder system module.

Table 18–1. altpll Advanced Signals

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 core are suitable for both synthesis and simulation. The PLL core supports the standard SOPC Builder simulation flow, so there are no special considerations for hardware simulation.

Register Definitions and Bit List

Table 18–2 shows the register map for the PLL core. Device drivers can control and communicate with the core through two 16-bit memory-mapped registers, `status` and `control`.

Note that the status and control bits shown below are present only if they have been created in the altpll MegaWizard, and set to **Register** on the **Interface** page in the PLL wizard.

Table 18–2. PLL Core Register Map

Offset	Register Name	R/W	Bit Description				
			15	…	2	1	0
0	`status`	R/O	*(1)*				LOCKED
1	`control`	R/W	*(1)*			PFDENA	ARESET

Note to Table 18–2:

(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

Status Register

Embedded software can access the PLL status via the `status` register. Writing to `status` has no effect. Table 18–3 describes the function of each bit.

Table 18–3. Status Register Bits

Bit Number	Bit Name	Value after reset	Description
0	LOCKED	1	Connects to the `locked` signal on the altpll. The `LOCKED` bit is high when valid clocks are present on the output of the PLL.
1 .. 15	—	—	Reserved. Read values are undefined.

Control Register

Embedded software can control the PLL via the `control` register. Software can also read back the status of control bits. Table 18–4 describes the function of each bit.

Table 18–4. Control Register Bits

Bit Number	Bit Name	Value after reset	Description
0	ARESET	0	Connects to the `areset` signal on the altpll. Writing a 1 to this bit asserts the `areset` signal for one clock cycle.
1	PFDENA	1	Connects to the `pfdena` signal on the altpll.
2 .. 15	—	—	Reserved. Read values are undefined. When writing, set reserved bits to zero.

Referenced Document

This chapter references the *altpll Megafunction User Guide*.

Document Revision History

Table 18–5 shows the revision history for this chapter.

Table 18–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 18 was formerly chapter 16.	—
May 2007 v7.1.0	● Chapter 16 was formerly chapter 14. ● Added table of contents to Overview section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies ● Changed old "Avalon switch fabric" term to "system interconnect fabric" ● Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface"	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	Initial release.	—

19. Performance Counter Core

QII55001-7.2.0

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® provides device drivers to enable the Nios II processor to use the performance counters.

This chapter contains the following sections:

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 a simple 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 19–1 shows the registers in detail.

Table 19–1. Performance Counter Core Register Map

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
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 19–1:

(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

System Reset Considerations

After system reset, the performance counter core is stopped and disabled, and all counters contain 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

Designers 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 19–2 lists macros and functions for accessing the performance counter hardware structure.

Table 19–2. 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.

Table 19–2. Performance Counter Macros and Functions

Name	Summary
`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 19–8.

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 19–3 lists the hardware constants.

Table 19–3. 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 19–3:
(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 19–4 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 situation 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.

For an example, refer to the performance checksum design files accompanying *AN 391: Profiling Nios II Systems*. These files may be found on the Altera Nios II literature page at www.altera.com/literature/lit-nio2.jsp.

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 19–1.

Example 19–1.

```
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 19–2 creates a table similar to this result.

Example 19–2.

```
--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 19–8.

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 function `perf_print_formatted_report()` from an ISR.

If a 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, ...)
```

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`.

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()`.

☞ This function requires the C standard library. Do not use the small C library with this function.

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 Document

This chapter references AN 391: Profiling Nios II Systems.

Document Revision History

Table 19–4 shows the revision history for this chapter.

Table 19–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	● Chapter 19 was formerly chapter 17 ● Removed incorrect statement about granularity of the timer.	—
May 2007 v7.1.0	● Chapter 17 was formerly chapter 15. ● Added table of contents to Overview section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	● Updated Avalon terminology because of changes to Avalon technologies ● Changed old “Avalon switch fabric” term to “system interconnect fabric” ● Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
December 2005 v5.1.0	Initial release.	—

Section VI. Streaming Peripherals

This section describes streaming peripherals provided by Altera® for SOPC Builder systems. These components allow you to optimize streaming applications.

Refer to *About This Handbook* for further details.

This section includes the following chapter:

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

20. Avalon Streaming Channel Multiplexer and Demultiplexer Cores

QII55004-7.2.0

Core Overview

The Avalon® streaming (Avalon-ST) channel multiplexer 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 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 demultiplexed 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 20–3
- "Demultiplexer" on page 20–6
- "Device and Tools Support" on page 20–8
- "Installation and Licensing" on page 20–9
- "Hardware Simulation Considerations" on page 20–9
- "Software Programming Model" on page 20–9

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 20–1 provides estimated resource utilization for eleven different configurations of the multiplexer.

Table 20–1. Multiplexer Estimated Resource Usage and Performance

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
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 20–2 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 20–2. 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

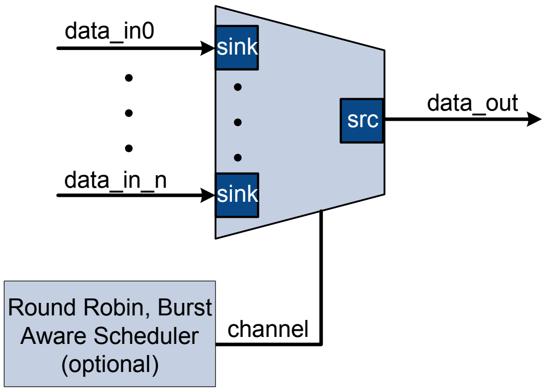
This section describes the hardware structure and functionality of the multiplexer component.

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 mux 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 mux is carrying data from a different input interface.

The mux includes an optional `channel` signal that enables each input interface to carry channelized data. When the `channel` signal is present on input interfaces, the mux 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 20–1).

Figure 20–1. 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 de-asserts `valid` on a ready cycle
- The 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—The following sections outline the options for the multiplexer as a whole:

- **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 high bits to indicate source port**—When selected, 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 mux has 4 input interfaces, then 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—The following sections outline the 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.

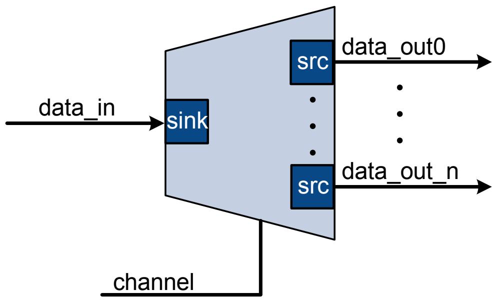
Demultiplexer

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 that each output interface will be idle when the demux is driving data to a different output interface. The demux 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 20–2).

Figure 20–2. The 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

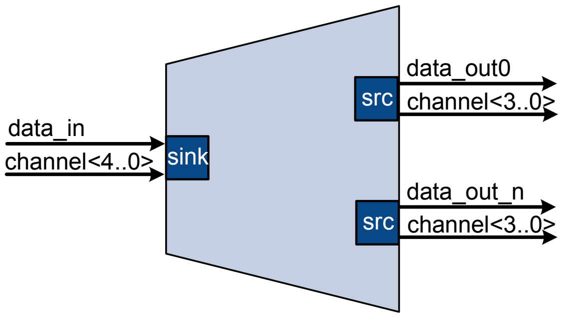
Use the MegaWizard interface for the demultiplexer core in SOPC Builder to specify the core configuration. The following sections list the available options in the MegaWizard interface.

Functional Parameters—The following sections outline the 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 selected, 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 not selected, 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 20–3 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 will go to channel 0 and the odd channels will go to channel 1. If the high-order bits of the channel signal select the output interface, channels 0–7 will go to channel 0 and channels 8–15 will go to channel 1.

Figure 20–3. Select Bits for Demultiplexer



Input Interface—The following sections outline the 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 – 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 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 and Tools Support

Altera device support for the multiplexer and demultiplexer components is listed in Table 20–3. For each device family, a component provides either full or preliminary support:

- *Full support* means the component meets all functional and timing requirements for the device family and may be used in production designs.
- *Preliminary support* means the component meets all functional requirements, but might still be undergoing timing analysis for the device family; it may be used in production designs with caution.

Table 20–3. Device Family Support

Device Family	Avalon-ST Multiplexer	Avalon-ST Demultiplexer
Arria™ GX	Preliminary	Preliminary
Cyclone III	Preliminary	Preliminary
Cyclone II	Full	Full
Cyclone	Full	Full
HardCopy® II	Full	Full
Stratix III	Preliminary	Preliminary
Stratix II GX	Full	Full
Stratix II	Full	Full
Stratix GX	Full	Full
Stratix	Full	Full

Installation and Licensing

The multiplexer and demultiplexer components are included in the Altera MegaCore® IP Library, which is an optional part of the Quartus® II software installation. After you install the MegaCore IP Library, SOPC Builder recognizes these components and can instantiate them into a system.

You can use the multiplexer and demultiplexer components for free without a license in any design targeting an Altera device.

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 demultiplexer at run-time. The components cannot generate interrupts.

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
October 2007 v7.2.0	Chapter 20 was formerly chapter 18.	—
May 2007 v7.1.0	Initial release.	—

21. Avalon Streaming Test Pattern Generator and Checker Cores

QII55007-7.2.0

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

Resource utilization and performance for the test pattern generator and checker cores depend on the datawidth, number of channels, and whether the streaming data uses the optional packet protocol.

Table 21–1 provides estimated resource utilization and performance for the test pattern generator core.

Table 21–1. 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 21–2 provides estimated resource utilization and performance for the test pattern checker core.

Table 21–2. 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

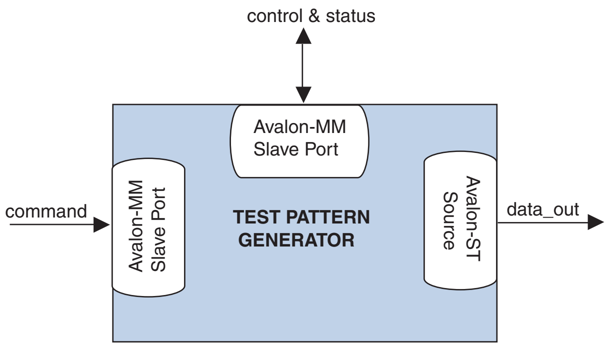
Test Pattern Generator

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 21–1 shows a block diagram of the test pattern generator core.

Figure 21–1. 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 wait request 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 21–12 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 1–256.
- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1–256. 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 1–256.
- **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 0–31. A value of 0 indicates that the `error` signal is not in use.

Test Pattern Checker

This section describes the hardware structure and functionality of the test pattern checker core.

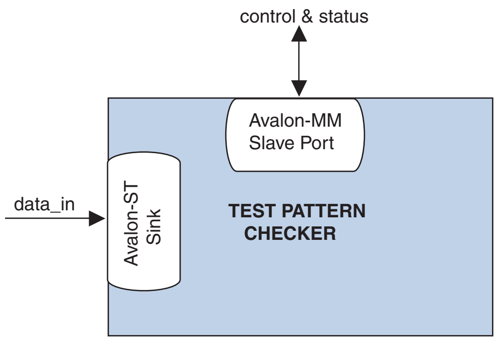
Functional Description

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 21–2 shows a block diagram of the test pattern checker core.

Figure 21–2. 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:

- **Number of Channels**—The number of channels that the test pattern checker core supports. Valid values are 1–256.
- **Data Bits Per Symbol**—The number of bits per symbol for the input interface. Valid values are 1–256.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat. 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.
- **Error Signal Width (bits)**—The width of the `error` signal on the input interface. Valid values are 0–31. A value of 0 indicates that the `error` signal is not in use.

Device and Tools Support

For each device family, the test pattern generator and checker cores provide either full or preliminary support:

- *Full support* means the component meets all functional and timing requirements for the device family and may be used in production designs.
- *Preliminary support* means the component meets all functional requirements, but might still be undergoing timing analysis for the device family; it may be used in production designs with caution.

Figure 21–3 shows the level of support offered by the test pattern generator and checker cores to each Altera device family.

Table 21–3. Device Family Support

Device Family	Support	
	Test Pattern Generator	Test Pattern Checker
Arria™ GX	Preliminary	Preliminary
Cyclone III	Preliminary	Preliminary
Cyclone II	Full	Full
Cyclone	Full	Full
HardCopy® II	Full	Full
Stratix III	Preliminary	Preliminary
Stratix II GX	Full	Full
Stratix II	Full	Full
Stratix GX	Full	Full
Stratix	Full	Full

Installation and Licensing

The test pattern generator and checker cores are included in the Altera MegaCore® IP Library, which is an optional part of the Quartus® II software installation. After you install the MegaCore IP Library, SOPC Builder recognizes these components and can instantiate them into a system.

You can use the test pattern generator and checker for free without a license in any design targeting an Altera device.

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**

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 21–4 shows the offset for the test pattern generator control and status registers. Each register is 32 bits wide.

Table 21–4. Test Pattern Generator Control and Status Register Map

Offset	Register Name
base + 0	`status`
base + 1	`control`
base + 2	`fill`

Table 21–5 describes the `status` register bits.

Table 21–5. 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 21–6 describes the `control` register bits.

Table 21–6. 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 and 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 and 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 21–7 describes the `fill` register bits.

Table 21–7. Fill Field Descriptions

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		
15:7	`FILL`	RO	The number of commands currently in the command FIFO.
31:16	Reserved		

Test Pattern Generator Command Registers

Table 21–8 shows the offset for the command registers. Each register is 32 bits wide.

Table 21–8. Test Pattern Generator Command Register Map

Offset	Register Name
base + 0	`cmd_lo`
base + 1	`cmd_hi`

Table 21–9 describes the `cmd_lo` register bits. The command is pushed into the FIFO only when the `cmd_lo` register is written to.

Table 21–9. 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 21–10 describes the `cmd_hi` register bits.

Table 21–10. 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 21–11 shows the offset for the control and status registers. Each register is 32 bits wide.

Table 21–11. 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 21–12 describes the `status` register bits.

Table 21–12. Status Field Descriptions (Part 1 of 2)

Bit(s)	Name	Access	Description
15:0	`ID`	RO	Contains a constant value of 0x65.
23:16	`NUMCHANNELS`	RO	The configured number of channels.

Table 21–12. Status Field Descriptions (Part 2 of 2)

Bit(s)	Name	Access	Description
30:24	`NUMSYMBOLS`	RO	The configured number of symbols per beat.
31	`SUPPORTPACKETS`	RO	A value of 1 indicates packet support.

Table 21–13 describes the `control` register bits.

Table 21–13. 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 and 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 and 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 21–14 describes the `exception_descriptor` register bits. If there is no exception, reading this register returns 0.

Table 21–14. 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 21–15 describes the `indirect_select` register bits.

Table 21–15. 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 21–16 describes the `indirect_count` register bits.

Table 21–16. 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 APIs 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.

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 and 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.

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_u32 channel, alt_u32 size, alt_u32 flags, alt_u32 error, alt_u32 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 XORed 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 APIs 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 and 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.

Referenced Document

This chapter references the *Avalon Streaming Interface Specification*.

Document Revision History

Table 21–17 shows the revision history for this chapter.

Table 21–17. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Initial release.	—