ALTERA®

Section II. Peripheral Support

This section provides information about the Nios® II peripherals.

This section includes the following chapters:

- Chapter 5, SDRAM Controller with Avalon Interface
- Chapter 6, DMA Controller with Avalon Interface
- Chapter 7, PIO Core With Avalon Interface
- Chapter 8, Timer Core with Avalon Interface
- Chapter 9, JTAG UART Core with Avalon Interface
- Chapter 10, UART Core with Avalon Interface
- Chapter 11, SPI Core with Avalon Interface
- Chapter 12, EPCS Device Controller Core with Avalon Interface
- Chapter 13, Common Flash Interface Controller Core with Avalon Interface
- Chapter 14, System ID Core with Avalon Interface
- Chapter 15, Character LCD (Optrex 16207) Controller with Avalon Interface
- Chapter 16, Mutex Core with Avalon Interface

Revision History

The table below shows the revision history for Chapters 5 – 16. These version numbers track the document revisions; they have no relationship to the version of the Nios II development kits or Nios II processor cores

Chapter(s)	Date / Version	Changes Made
5	September 2004, v1.1	Updates for Nios II 1.01 release.
	May 2004, v1.0	First publication.
6	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	First publication.
7	September 2004, v1.1	Updates for Nios II 1.01 release.
	May 2004, v1.0	First publication.
8	September 2004, v1.1	Updates for Nios II 1.01 release.
	May 2004, v1.0	First publication.
9	December 2004, v1.2	Added Cyclone II support.
	September 2004, v1.1	Updates for Nios II 1.01 release.
	May 2004, v1.0	First publication.
10	September 2004, v1.1	Updates for Nios II 1.01 release.
	May 2004, v1.0	First publication.
11	September 2004, v1.1	Updates for Nios II 1.01 release.
	May 2004, v1.0	First publication.
12	September 2004, v1.1	Updates for Nios II 1.01 release.
	May 2004, v1.0	First publication.
13	December 2004, v1.2	Added Cyclone II support.
	September 2004, v1.1	Updates for Nios II 1.01 release.
	May 2004, v1.0	First publication.
14	September 2004, v1.1	Updates for Nios II 1.01 release.
	May 2004, v1.0	First publication.

Chapter(s)	Date / Version	Changes Made
15	September 2004, v1.0	First publication.
16	December 2004, v1.0	First publication.

5. SDRAM Controller with Avalon Interface

NII51005-1.1

Core Overview

The SDRAM controller with Avalon™ interface provides an Avalon 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 slave port that appears as linear memory (i.e., flat address space) to Avalon master peripherals.

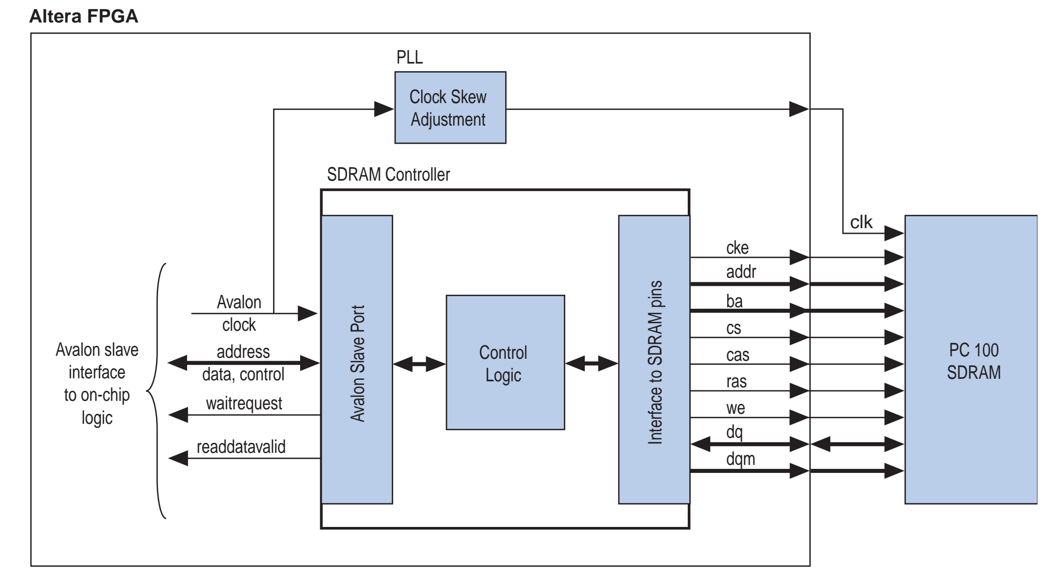
The core can access SDRAM subsystems with various data widths (8, 16, 32, or 64 bits), various memory sizes, and multiple chip selects. The Avalon interface is latency-aware, allowing read transfers to be pipelined. The core can optionally share its address and data buses with other off-chip Avalon tristate 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 with Avalon Interface is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

Functional Description

Figure 5–1 shows a block diagram of the SDRAM controller core connected to an external SDRAM chip.

Figure 5–1. 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 run-time.

Avalon Interface

The Avalon slave port is the only 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 interface behaves as a simple memory interface. There are no memory-mapped configuration registers.

The Avalon 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.

See the *Avalon Interface Specification Reference Manual* for details on Avalon transfer types.

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) via I/O pins on the Altera FPGA.

Signal Timing & 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 5–6 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.

Synchronization

The SDRAM chip is driven at the same clock rate as the Avalon interface. As shown in Figure 5–1, an on-chip phase-locked loop (PLL) is often used to alleviate clock skew between the SDRAM controller core and the SDRAM chip. At lower clock speeds, the PLL may not be necessary. At higher clock rates, a PLL becomes necessary to tune the SDRAM clock to toggle within the window when signals are valid on the pins.

The PLL block is not an integral part of the SDRAM controller core. If the PLL is necessary, the designer must manually instantiate the PLL outside the SOPC Builder-generated system module. Different combinations of Altera FPGA and SDRAM chip will require different PLL settings.

The SDRAM controller does not support clock-disable modes. The SDRAM controller permanently asserts the `cke` pin.

The Nios® II development kit provides an example hardware design that uses the SDRAM controller core in conjunction with a PLL.

Sharing Pins with Other Avalon Tristate Devices

If an Avalon tristate bridge is present in the SOPC Builder system, the SDRAM controller core can share pins with the existing tristate bridge. In this case, the core's `addr`, `dq` (data) and `dqm` (byte-enable) pins are shared with other devices connected to the Avalon tristate bridge. This feature

conserves I/O pins, which is valuable in systems that have multiple external memory chips (e.g., flash, SRAM, in addition to SDRAM), but too few pins to dedicate to the SDRAM chip. See "Performance Considerations" on page 5–4 for details on how pin sharing affects performance.

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, wherein 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 will operate at rates approaching one word per clock. Applications that frequently access different destination banks will require extra management cycles for row closings and openings.

Sharing Data & Address Pins

When the controller shares pins with other tristate devices, average access time usually increases while bandwidth decreases. When access to the tristate 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 tristate bridge as long as back-to-back read or write transactions continue within the same row and bank.

Note that this behavior may degrade the average access time for other devices sharing the Avalon tristate 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 tristate bridge.
- The controller is guaranteed not to violate the SDRAM's row open time limit.

Hardware Design & 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® brand FPGAs. However, the core does not guarantee 100 MHz performance in all Altera FPGA families.

The f_{MAX} performance also depends on the overall hardware design. The master clock for the SOPC Builder system module drives both the SDRAM controller core and the SDRAM chip. Therefore, the overall system module's performance determines the performance of the SDRAM controller. For example, to achieve f_{MAX} performance of 100 MHz, the system module must be designed for a 100-MHz clock rate, and timing analysis in the Quartus II software must verify that the hardware design is capable of 100-MHz operation.

Device & 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 on supported I/O types, see the handbook for the target FPGA family.

Instantiating the Core in SOPC Builder

Designers use the configuration wizard for the SDRAM controller in SOPC Builder to specify hardware features and simulation features. The SDRAM controller configuration wizard has two tabs: **Memory Profile** and **Timing**. This section describes the options available on each tab.

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, then the SDRAM controller core can be configured easily by selecting the appropriate preset value. The following preset configurations are defined:

- Micron MT8LSDT1664HG module
- Four SDR100 8 MByte x 16 chips
- Single Micron MT48LC2M32B2-7 chip
- Single Micron MT48LC4M32B2-7 chip
- Single NEC D4564163-A80 chip (64 MByte x 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 tab changes the **Preset** value to **custom**.

Memory Profile Tab

The **Memory Profile** tab 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 5–1 lists the settings available on the **Memory Profile** tab.

Table 5–1. Memory Profile Tab Settings

<table>
<tr><th colspan="2">Settings</th><th>Allowed Values</th><th>Default Values</th><th>Description</th></tr>
<tr><td colspan="2">Data Width</td><td>8, 16, 32, 64</td><td>32</td><td>SDRAM data bus width. This value determines the width of the dq bus (data) and the dqm bus (byte-enable).</td></tr>
<tr><td rowspan="2">Architecture Settings</td><td>Chip Selects</td><td>1, 2, 4, 8</td><td>1</td><td>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.</td></tr>
<tr><td>Banks</td><td>2, 4</td><td>4</td><td>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.</td></tr>
<tr><td rowspan="2">Address Width Settings</td><td>Row</td><td>11, 12, 13, 14</td><td>12</td><td>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.</td></tr>
<tr><td>Column</td><td>>= 8, and less than Row value</td><td>8</td><td>Number of column address bits. For example, the SDRAM organized as 4096 rows by 512 (2^9) columns has a Column value of 9.</td></tr>
<tr><td colspan="2">Controller shares dq/dqm/addr I/O pins</td><td>Yes, No</td><td>No</td><td>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, SOPC Builder presents a new configuration tab that allows the user to associate the SDRAM controller pins with a specific tristate bridge.</td></tr>
<tr><td colspan="2">Include a functional memory model in the system testbench</td><td>Yes, No</td><td>Yes</td><td>When this option is turned 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 5–9.</td></tr>
</table>

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

Timing Tab

The **Timing** tab allows designers to enter the timing specifications of the SDRAM chip(s) used. The correct values are provided in the manufacturer's data sheet for the target SDRAM. Table 5–2 lists the settings available on the **Timing** tab.

Table 5–2. Timing Tab Settings

Settings	Allowed Values	Default Values	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 will perform 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 met 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 input by the user, the actual timing achieved for each parameter will be integer multiples of the Avalon clock. For the **Issue one refresh command every** parameter, the actual timing will be 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. There are three major components required for simulation:

- The simulation model for the SDRAM controller
- The 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. There is nothing ModelSim-specific about the SDRAM controller simulation model. However, minor changes may be required to make the model work with other simulators.

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

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

SDRAM Memory Model

There are two options for simulating a memory model of the SDRAM chip(s), as described below.

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 test bench.

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 5–2 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 5–2. Single 128-Mbit SDRAM Chip with 32-Bit Data

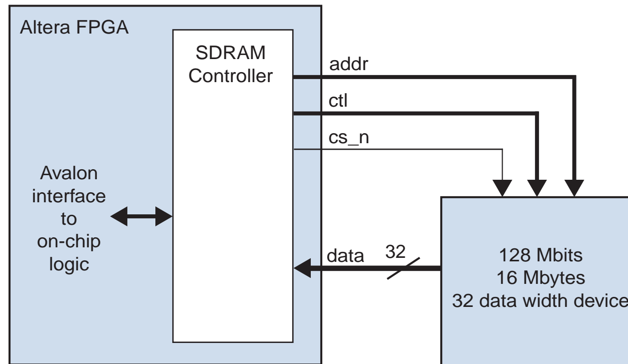


Figure 5–3 shows two 64-Mbit SDRAM chips, each with 16-bit data. Address and control signals wire in parallel to both chips. Note that chipselect (`cs_n`) is shared by the chips. Each chip provides half of the 32-bit data bus. The result is a logical 128-Mbit (16-Mbyte) 32-bit data memory.

Figure 5–3. Two 64-MBit SDRAM Chips Each with 16-Bit Data

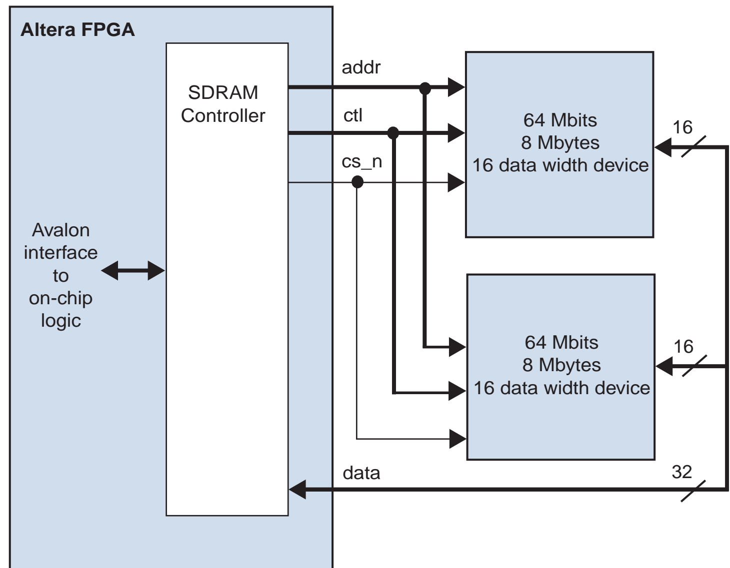
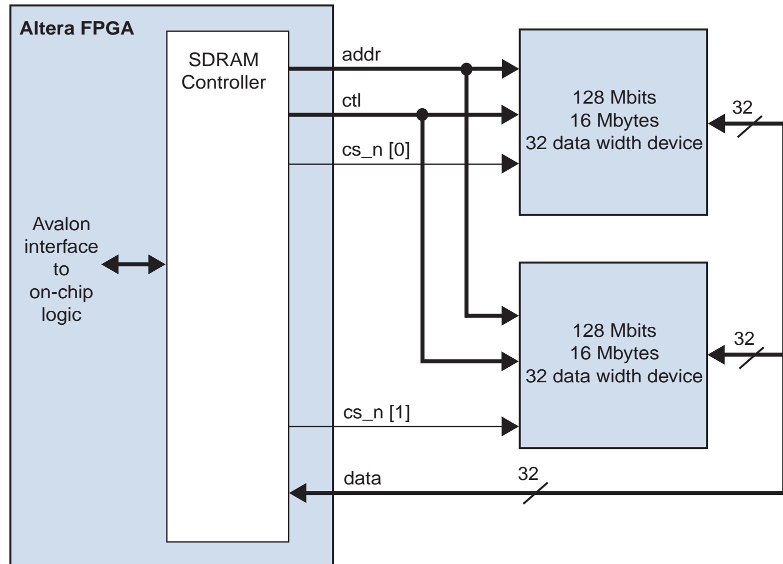


Figure 5–4 shows two 128-Mbit SDRAM chips, each with 32-bit data. Control, address and data signals wire in parallel to the two chips. The chipselect bus (`cs_n[1:0]`) determines which chip is selected. The result is a logical 256-Mbit 32-bit wide memory.

Figure 5–4. Two 128-Mbit SDRAM Chips Each with 32-Bit Data



Software Programming Model

The SDRAM controller behaves like simple memory when accessed via the Avalon 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.

6. DMA Controller with Avalon Interface

NII51006-1.2

Core Overview

The Direct Memory Access (DMA) controller with Avalon™ interface ("the DMA controller") performs bulk data transfers, reading data from a source address range and writing the data to a different address range. An Avalon 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 streaming Avalon transfers, enabling it to automatically transfer data to or from a slow streaming peripheral (e.g., a universal asynchronous receiver/transmitter [UART]), at the maximum pace allowed by the peripheral.

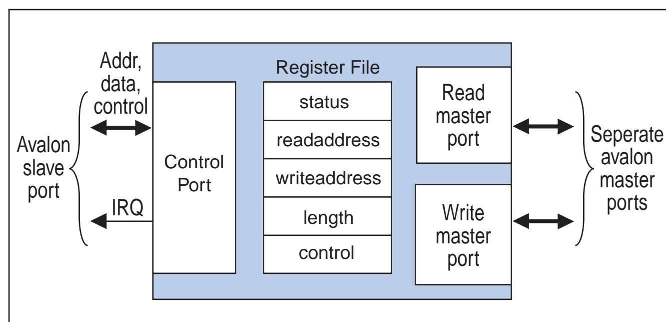
The DMA controller is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios® II processor, device drivers are provided in the HAL system library. See "Software Programming Model" on page 6–5 for details of HAL support.

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 slave peripheral (i.e., a constant address) or an address range in memory. The DMA controller can be used in conjunction with streaming-capable peripherals, which allows data transactions of fixed or variable length. The DMA controller can signal an interrupt request (IRQ) when a DMA transaction completes. This document defines a transaction as a sequence of one or more Avalon transfers initiated by the DMA controller core.

The DMA controller has two Avalon master ports—a master read port and a master write port—and one Avalon slave port for controlling the DMA as shown in Figure 6–1.

Figure 6–1. X. 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 (i.e., 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 master peripheral sets up and initiates DMA transactions by writing to registers via the control port. 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 & 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. 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 streaming transfers, which allows the slave peripheral to control the flow of data and terminate the DMA transaction.

For details on streaming Avalon data transfers and streaming Avalon peripherals, see the *Avalon Interface Specification Reference Manual*.

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 6–1.

Table 6–1. Address Increment Values

Transfer Width	Increment
byte	1
halfword	2
word	4
doubleword	8
quadword	16

Instantiating the Core in SOPC Builder

Designers use the DMA controller's SOPC Builder configuration wizard to specify hardware options for the target system. Instantiating the DMA controller in SOPC Builder creates one slave port and two master ports. The designer must specify which slave peripherals can be accessed by the read and write master ports. Likewise, the designer 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.

The configurable hardware features are described below.

DMA Parameters (Basic)

The following sections describe the basic parameters.

Width of the DMA Length Register

This option sets the minimum width of the DMA's transaction length register. The acceptable range is 1 to 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.

Construct FIFO from Registers vs. Construct FIFO from Memory Blocks

This option controls the implementation of the FIFO buffer between the master read and write ports. When **Construct FIFO from Registers** is selected (the default), the FIFO is implemented 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 6–5). When **Construct FIFO from Memory Blocks** is selected, the FIFO is implemented using embedded memory blocks available in the FPGA.

Advanced Options

This section describes the advanced options.

Allowed Transactions

The designer can choose the transfer data width(s) supported by the DMA controller hardware. The following data-width 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.

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.

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 6–2 lists the available operations. These are valid for both the transmit and receive channels.

Table 6–2. Operations for alt_dma_rxchan_ioctl() & 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 6–2:

(1) These macro names changed in version 1.1 of the Nios II development kit. 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.

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

Table 6–3 shows the register map for the DMA controller. Device drivers control and communicate with the hardware through five memory-mapped 32-bit registers.

Table 6–3. DMA Controller Register Map

Offset	Register Name	Read/Write	31. . .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											

Table 6–3. DMA Controller Register Map

Offset	Register Name	Read/Write	31. . .11	10	9	8	7	6	5	4	3	2	1	0
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)* *(4)*	*(5)*	WCON	RCON	LEEN	WEEN	REEN	I_EN	GO	WORD	HW	BYTE
7		-	Reserved *(3)*											

Notes:
(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.
(4) QUADWORD.
(5) DOUBLEWORD.

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 6–4.

Table 6–4. 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 6–5.

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

Bit Number	Bit Name	Read/Write/Clear	Description
0	BYTE	RW	Specifies byte transfers.
1	HW	RW	Specifies halfword (16-bit) transfers.
2	WORD	RW	Specifies word (32-bit) transfers.

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

Bit Number	Bit Name	Read/Write/Clear	Description
3	GO	RW	Enables DMA transaction. When the GO bit is set to 0, the DMA is prevented from executing transfers. When the GO bit is set to 1 and the length register is non-zero, transfers occur.
4	I_EN	RW	Enables interrupt requests (IRQ). When the I_EN bit is 1, the DMA controller generates an IRQ when the status register's DONE bit is set to 1. IRQs are disabled when the I_EN bit is 0.
5	REEN	RW	Ends transaction on read-side end-of-packet. When the REEN bit is set to 1, a streaming slave port 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 streaming slave port on the write side may end the DMA transaction by asserting its end-of-packet signal.
7	LEEN	RW	Ends transaction when the `length` register reaches zero. When the LEEN bit is 1, the DMA transaction ends when the `length` register reaches 0. When this bit is 0, `length` reaching 0 does not cause a transaction to end. In this case, the DMA transaction must be terminated by an end-of-packet signal from either the read or write master port.
8	RCON	RW	Reads from a constant address. When RCON is 0, the read address increments after every data transfer. This is the mechanism for the DMA controller to read a range of memory addresses. When RCON is 1, the read address does not increment. This is the mechanism for the DMA controller to read from a peripheral at a constant memory address. For details, see "Address Incrementing" on page 6–3.
9	WCON	RW	Writes to a constant address. Similar to the RCON bit, when WCON is 0 the write address increments after every data transfer; when WCON is 1 the write address does not increment. For details, see "Address Incrementing" on page 6–3.
10	DOUBLEWORD	RW	Specifies doubleword transfers.
11	QUADWORD	RW	Specifies quadword transfers.

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.

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.

7. PIO Core With Avalon Interface

NII51007-1.1

Core Overview

The parallel input/output (PIO) core provides a memory-mapped interface between an Avalon™ 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:

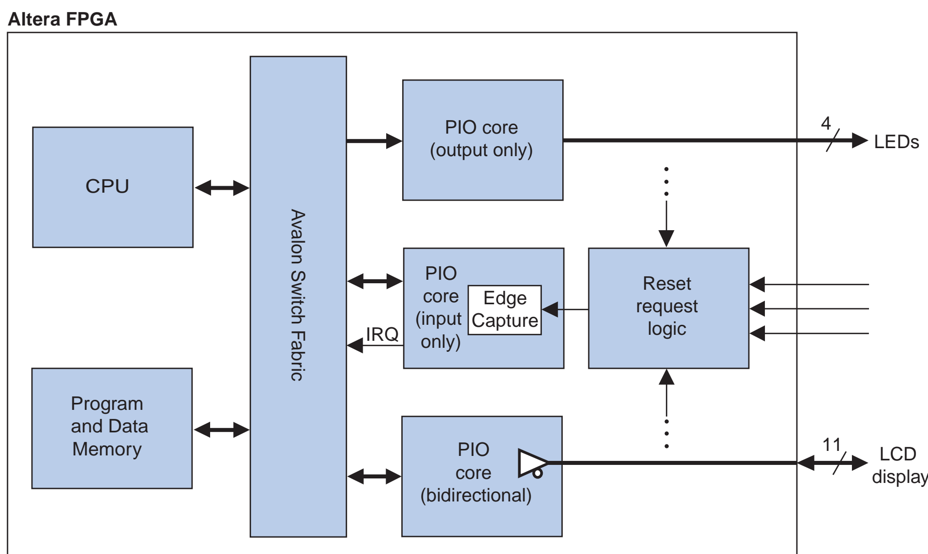
- Controlling LEDs
- Acquiring data from switches
- Controlling display devices
- Configuring and communicating with off-chip devices, such as application-specific standard products (ASSP)

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.

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 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 7–1 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 7–1. 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 interface. See Table 7–2 on page 7–7 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 & 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 7–2 shows a block diagram of the PIO core configured with input and output ports, as well as support for IRQs.

Figure 7–2. PIO Core with Input & Output Ports & with IRQ Support

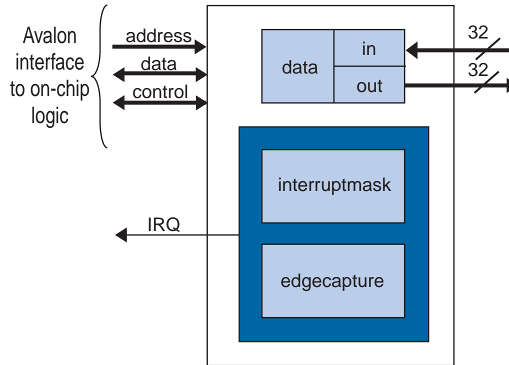
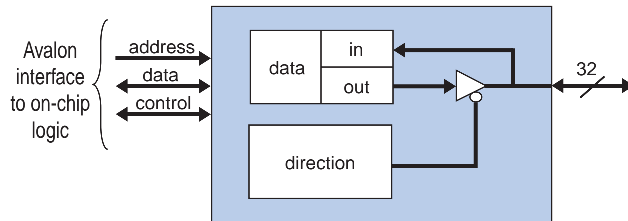


Figure 7–3 shows a block diagram of the PIO core configured in bidirectional mode, without support for IRQs.

Figure 7–3. PIO Core with Bidirectional Ports



Avalon Interface

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

Instantiating the PIO Core in SOPC Builder

The hardware feature set is configured via the PIO core's SOPC Builder configuration wizard. The following sections describe the available options.

The configuration wizard has two tabs, **Basic Settings** and **Input Options**.

Basic Settings

The **Basic Settings** tab 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 7–1.

Table 7–1. 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** tab allows the designer to specify edge-capture and IRQ generation settings. The **Input Options** tab is not available when **Output ports only** is selected on the **Basic Settings** tab.

Edge Capture Register

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

Interrupt

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

Device & 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 Development Kit 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 on these routines, refer to the PIO documentation that accompanied the first-generation Nios processor. For details on 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 master peripheral, such as a CPU, controls and communicates with the PIO core via the four 32-bit registers, shown in Table 7–2. The table assumes that the PIO core's I/O ports are configured to a width of *n* bits.

Table 7–2. 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 7–2:

(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.

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 7–9.

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 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 interrupt-request (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.

8. Timer Core with Avalon Interface

NII51008-1.1

Core Overview

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

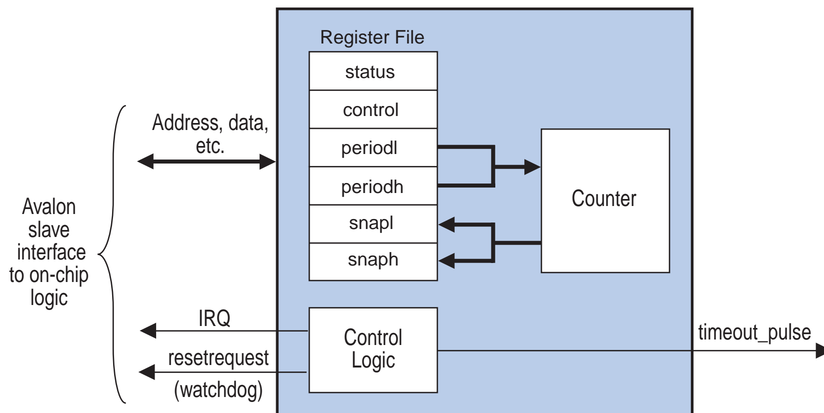
- Controls to start, stop, and reset the timer
- Two count modes: count down once and continuous count-down
- Count-down period register
- Maskable interrupt request (IRQ) upon reaching zero
- Optional watchdog timer feature that resets the system if timer ever reaches zero
- Optional periodic pulse 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.

Functional Description

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

Figure 8–1. Timer Core Block Diagram



The timer core has two user-visible features:

- The Avalon 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 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 Slave Interface

The timer core implements a simple Avalon slave interface to provide access to the register file. The Avalon 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 peripherals in the SOPC Builder system. When the `resetrequest` signal is asserted, it forces any processor connected to the system to reboot. See "Configuring the Timer as a Watchdog Timer" on page 8–4 for further details.

Device & Tools Support

The timer core supports all Altera® FPGA families.

Instantiating the Core in SOPC Builder

Designers use the timer's SOPC Builder configuration wizard to specify the hardware features. This section describes the options available in the configuration wizard.

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 the **Writeable period** setting (see below) is turned off, the period is fixed and cannot be updated at runtime.

The **Timeout Period** setting can be specified in units of **usec**, **msec**, **sec**, or **clocks** (number of clock cycles). The actual period achieved depends on the system clock. If the period is specified in usec, msec or sec, the true period will be the smallest number of clock cycles that is greater than or equal to the specified **Timeout Period**.

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. See "Configuring the Timer as a Watchdog Timer" on page 8–4.

Register Options

Table 8–1 shows the settings that affect the timer core's registers.

Table 8–1. 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 8–2 shows the settings that affect the timer core's output signals.

Table 8–2. 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 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. See "Configuring the Timer as a Watchdog Timer" on page 8–4.

Configuring the Timer as a Watchdog Timer

To configure the timer for use as a watchdog, in the configuration wizard select **Watchdog** in the **Preset Configurations** list, or choose the following settings:

- Set the **Timeout Period** to the desired "watchdog" period.
- Turn **off** the **Writeable period** option.
- Turn **off** the **Readable snapshot** option.

- Turn **off** the **Start/Stop control bits** option.
- Turn **off** the **Timeout pulse** option.
- Turn **on** the **System reset on timeout (watchdog)** option.

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 `snapshot` 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.

See the *Nios II Software Developer's Handbook* for details on using the system clock and timestamp features that use these drivers. The Nios II development kit 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.

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 8–3 shows the register map for the timer.

Table 8–3. Register Map

Offset	Name	R/W	Description of Bits						
			15	…	4	3	2	1	0
0	`status`	RW	*(1)*					RUN	TO
1	`control`	RW	*(1)*			STOP	START	CONT	ITO
2	`periodl`	RW	Timeout Period – 1 (bits 15..0)						
3	`periodh`	RW	Timeout Period – 1 (bits 31..16)						
4	`snapl`	RW	Counter Snapshot (bits 15..0)						
5	`snaph`	RW	Counter Snapshot (31..16)						

Note to Table 8–3:

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

status Register

The `status` register has two defined bits, as shown in Table 8–4.

Table 8–4. 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 8–5.

Table 8–5. control Register Bits

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.
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 the **Start/Stop control bits** option turned off, writing the STOP bit has no effect.

Note:
(1) Writing 1 to both START and STOP bits simultaneously produces an undefined result.

periodl & 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 (0x00000000) for one clock cycle.

Writing to either `periodh` or `periodl` stops the internal counter, except when the hardware is configured with the **Start/Stop control bits** option turned off. If the **Start/Stop control bits** option is turned off, writing either register does not stop the counter. When the hardware is configured with the **Writeable period** option disabled, writing to either `periodh` or `periodl` causes the counter to reset to the fixed **Timeout Period** specified at system generation time.

snapl & 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

9. JTAG UART Core with Avalon Interface

NII51009-1.2

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.

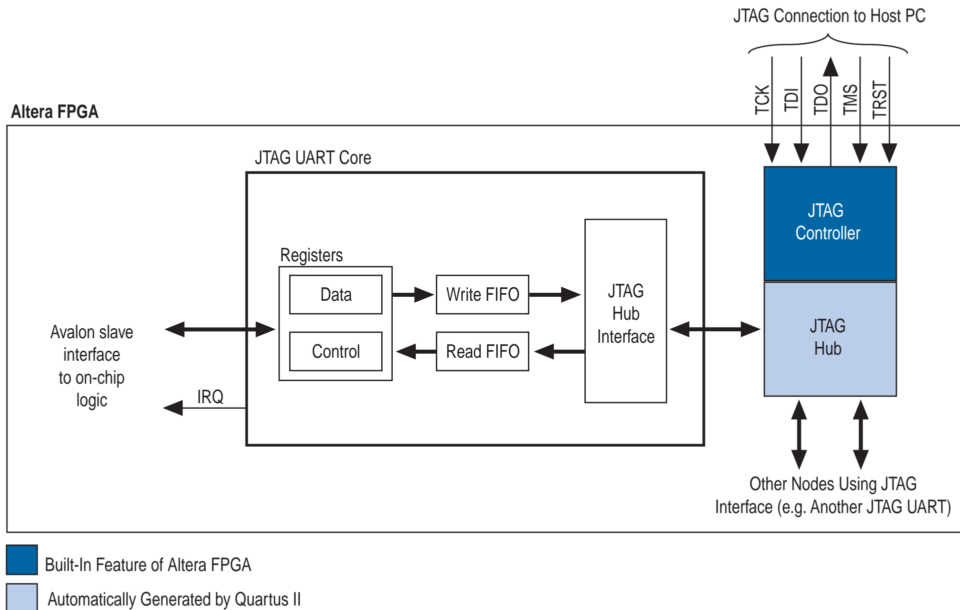
The JTAG UART core uses the JTAG circuitry built in to Altera FPGAs, and provides host access via the JTAG pins on the FPGA. The host PC can connect to the FPGA via any Altera JTAG download cable, such as the USB-Blaster™ cable. Software support for the JTAG UART core is provided by Altera. For the Nios 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.

Functional Description

Figure 9–1 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 9–1. JTAG UART Core Block Diagram



Avalon Slave Interface & 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 9–13.

Read & 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 designers to trade off logic resources for memory resources, if necessary.

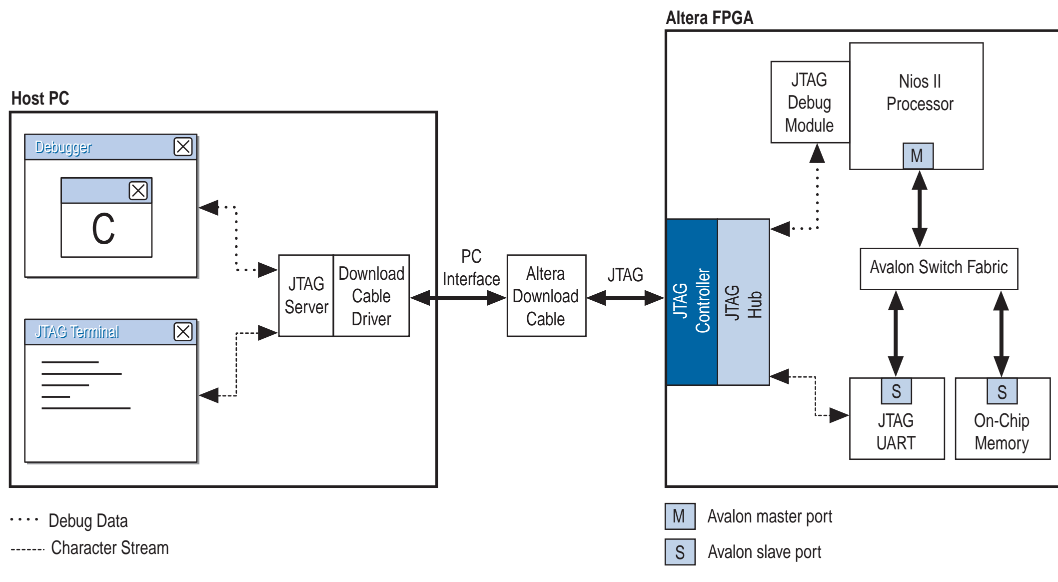
JTAG Interface

Altera FPGAs contain built-in JTAG control circuitry that interfaces the device's JTAG pins to logic inside the device. The JTAG controller can connect to user-defined circuits called "nodes" implemented in the FPGA. Because there may be several nodes that need to communicate via the JTAG interface, a JTAG hub (i.e., a multiplexer) becomes 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; it is presented here only for clarity.

Host-Target Connection

Figure 9–2 shows the connection between a host PC and an SOPC Builder-generated system containing a JTAG UART core.

Figure 9–2. 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 9–2 contains one JTAG UART core and a Nios II processor. Both agents communicate to 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. Only one processor should communicate with each JTAG UART core to maintain coherent data streams.

Device Support & Tools

The JTAG UART core supports the Stratix®, Stratix II, Cyclone™ and Cyclone II 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 JTAG UART core's SOPC Builder configuration wizard to specify the core features. The following sections describe the available options in the configuration wizard.

Configuration Tab

The options on this tab 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 allowable. 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 efficiency, 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 9–13 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 acceptable. 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 efficiency, 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 9–13 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 configuration wizard 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 of simulating the JTAG UART core in Nios II systems.

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

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 precaution so that 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.

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()`.

"Printing Characters to a JTAG UART Core as stdout" 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 has been configured to use this JTAG UART device for stdout.

Printing Characters to a JTAG UART Core as stdout

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

"Transmitting Characters to a JTAG UART Core" on page 9–9 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.

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 development kit provides a number of software example designs that use the JTAG UART core.

Driver Options: Fast vs. Small Implementations

To accommodate the requirements of different types of systems, the JTAG UART driver provides two variants: A fast version and a small version. The fast behavior will be 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 there is no host 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 as well.
- Specify the preprocessor option `-DALTERA_AVALON_JTAG_UART_SMALL`. You can 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 9–1.

Table 9–1. 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.

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

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.

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 9–2 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 9–2. 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 9–2:

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

Data Register

Embedded software accesses the read and write FIFOs via the `data` register. Table 9–3 describes the function of each bit.

Table 9–3. 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 is a character to be written to the write FIFO. When reading, the DATA field is 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, else DATA is undefined.
16 .. 32	RAVAIL	R	The number of characters remaining in the read FIFO (after this 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 9–4 describes the function of each bit.

Table 9–4. 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 what condition generated the IRQ. See "Interrupt Behavior" on page 9–13 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 are pending and enabled.

Interrupt behavior is of concern to device driver programmers concerned with the bandwidth performance to the host PC. Example designs and the JTAG terminal program provided with Nios II development kits 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 will be 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.

10. UART Core with Avalon Interface

NII51010-1.1

Core Overview

The universal asynchronous receiver/transmitter core with Avalon™ interface ("the 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.

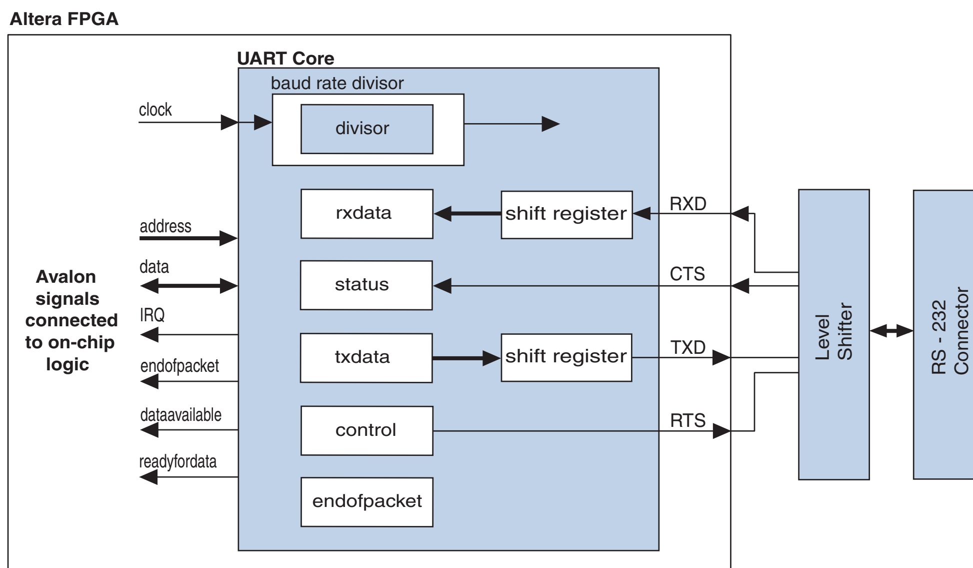
The core provides a simple register-mapped Avalon slave interface that allows Avalon master peripherals (such as a Nios® II processor) to communicate with the core simply by reading and writing control and data registers.

The UART core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

Functional Description

Figure 10–1 shows a block diagram of the UART core.

Figure 10–1. 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 slave port
- The RS-232 signals, `RXD`, `TXD`, `CTS`, and `RTS`

Avalon Slave Interface & Registers

The UART core provides an Avalon 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 see "Interrupt Behavior" on page 10–20.

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

See Chapter 6, DMA Controller with Avalon Interface for details. See the *Avalon Interface Specification Reference Manual* for details of the Avalon interface.

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 (e.g., 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 master peripherals write the `txdata` holding register via the Avalon slave port. The transmit shift register is automatically loaded from the `txdata` register 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 master peripherals read the `rxdata` holding register via the Avalon 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 clock input. The internal baud clock is generated by a clock divider. The divisor value can come from one of the following sources:

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

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

Device Support & Tools

The UART core can target all Altera FPGAs, including Stratix™ and Cyclone™ device families.

Instantiating the Core in SOPC Builder

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

The hardware feature set is configured via the UART core's SOPC Builder configuration wizard. 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 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 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 (e.g., 9600, 57600, 115200 bps), or you can manually enter any baud rate.

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:

$$divisor = int(\ (clock\ frequency)/(baud\ rate) + 0.5\ )$$

$$baud\ rate = (clock\ frequency)/(divisor + 1)$$

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 writeable, 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
See Table 10–1.

Table 10–1. Data Bits Setting

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. See below for further details.

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 pe (parity error) 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 1 if the character has an even number of 1 bits; otherwise the parity bit is 0. Similarly, when parity is Odd, the parity bit is 1 if the character has an odd number of 1 bits.

Flow Control

The following flow control option is available.

Include CTS/RTS pins & 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 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 the **Include CTS/RTS pins and control register bits** setting is off, the core does not include the hardware listed above. The control/status bits CTS, DCTS, IDCTS, and RTS are not implemented; they always read as 0.

Streaming Data (DMA) Control

The UART core's Avalon interface optionally implements streaming Avalon transfers. This allows an Avalon 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 interface to support streaming data transfers to/from other master peripherals in the system

End-of-packet (EOP) detection allows the UART core to terminate a streaming data transaction with a streaming-capable Avalon master. 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 (end of packet) character's value is determined by the `endofpacket` register.

When the end-of-packet 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 will be simulated entering the `RXD` port upon simulated system reset. The UART core's configuration wizard 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 on simulating the UART core in Nios II processor systems see *AN 351: Simulating Nios II Processor Designs*. For details on simulating the UART core in Nios embedded processor systems see *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.

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. See "Driver Options: Fast vs. Small Implementations" on page 10–11.

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: 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: Sending & 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;
}
```

The *Nios II Software Developer's Handbook* provides complete details of the HAL system library.

Driver Options: Fast vs. 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 behavior will be 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 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 you do not want to affect the drivers for other devices.

See the help system in the Nios II IDE for details on 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 10–2 defines operation requests that the UART driver supports.

Table 10–2. 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. See the comments above for details. The `ioctl` "arg" parameter is ignored.

Additional operation requests are also optionally available for the fast driver only, as shown in Table 10–3. To enable these operations in your program, you must set the preprocessor option `-DALTERA_AVALON_UART_USE_IOCTL`.

Table 10–3. 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 10–3:

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

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

Limitations

The HAL driver for the UART core does not support the endofpacket register. See "Register Map" on page 10–13 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 on these routines, refer to the UART documentation that accompanied the first-generation Nios processor. For details on 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 10–4 shows the register map for the UART core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 10–4. 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	`endofpacket` *(4)*	RW	*(1)*					*(2)*	*(2)*	End-of-Packet Value						

Notes to Table 10–4:

(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 below.

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 (i.e., 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 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 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 10–5.

Table 10–5. 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. See "Data Bits, Stop Bits, Parity" on page 10–6.
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 (i.e., 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 (i.e., 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 (i.e., a character is not being transmitted) the TMT bit is 1. An Avalon 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 10–5. 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 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 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 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. See “Flow Control” on page 10–6.
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 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. See “Flow Control” on page 10–6.

Table 10–5. 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. See “Streaming Data (DMA) Control” on page 10–7.

Note to Table 10–5:
(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 10–6.

Table 10–6. control Register Bits

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.
6	ITRDY	RW	Enable interrupt for a transmission ready.

Table 10–6. control Register Bits

Bit	Bit Name	Read/ Write	Description
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 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 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 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. See "Flow Control" on page 10–6.
12	IEOP	RW	Enable interrupt for end-of-packet condition.

Note to Table 10–6:

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

divisor Register (Optional)

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

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

The `divisor` register is an optional hardware feature. If the **Baud Rate Can Be Changed By Software** hardware option is not enabled, 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 see "Baud Rate Options" on page 10–5.

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, see Table 10–5 on page 10–16 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 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 10–5 on page 10–16 and Table 10–6 on page 10–18. Details of each interrupt condition are provided in the status bit descriptions.

11. SPI Core with Avalon Interface

NII51011-1.1

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 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 (e.g., 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.

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

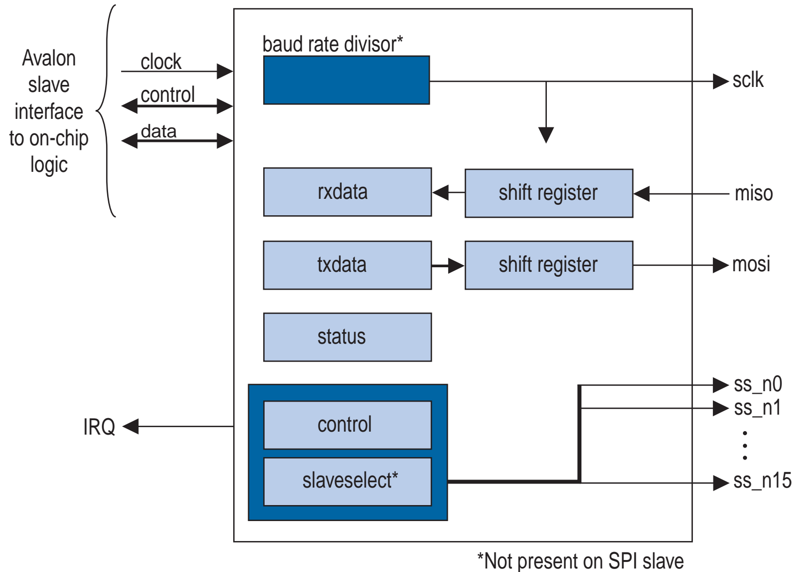
The SPI core has the following user-visible features:

- 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 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 11–1 shows a block diagram of the SPI core in master mode.

Figure 11–1. SPI Core Block Diagram

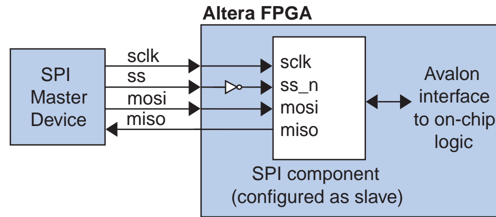


The SPI core logic is synchronous to the clock input provided by the Avalon interface. When configured as a master, the core divides the Avalon clock to generate the SCLK output. When configured as a slave, the core's receive logic is synchronized to SCLK input. The core's Avalon interface is capable of streaming Avalon transfers. The SPI core can be used in conjunction with a streaming DMA controller to automate continuous data transfers between, for example, the SPI core and memory. See Chapter 6, DMA Controller with Avalon Interface for details.

Example Configurations

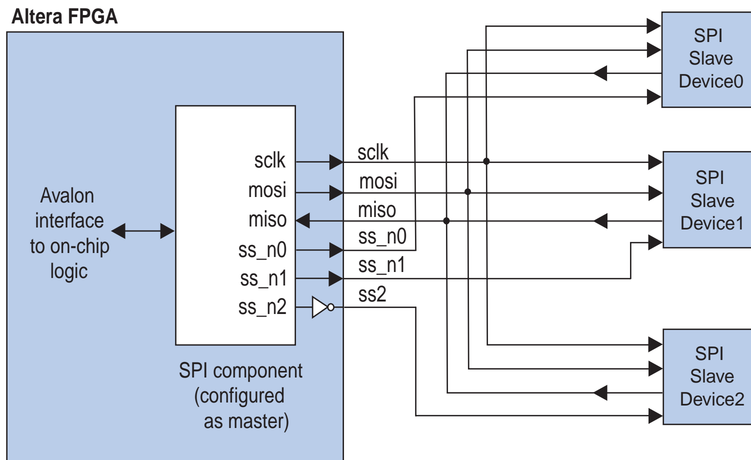
Two possible configurations are shown below. In Figure 11–2, the SPI core provides a slave interface to an off-chip SPI master.

Figure 11–2. SPI Core Configured as a Slave



In Figure 11–3 the SPI core provides a master interface driving multiple off-chip slave devices. Each slave device in Figure 11–3 must tristate its `miso` output whenever its select signal is not asserted.

Figure 11–3. 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 & 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 11–1.

Table 11–1. 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 (e.g., 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 (i.e., 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 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 11–2.

Table 11–2. 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

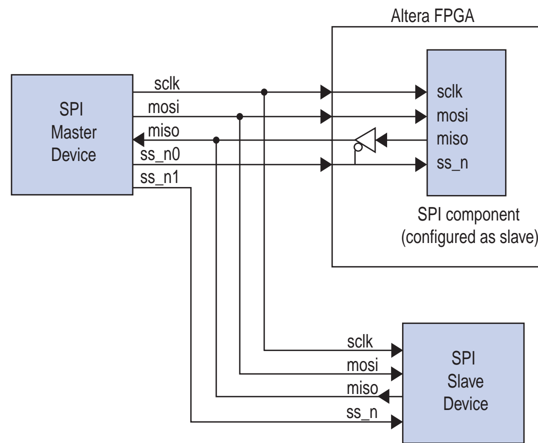
In slave mode, the SPI core simply waits for the master to initiate transactions. Before a transaction begins, the slave logic is continuously polling the ss_n input. When the master asserts ss_n (drives it low), the slave logic immediately begins sending the transmit shift register contents to the miso 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 (e.g., 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 11–4 shows an example of the SPI core in slave mode in an environment with two slaves.

Figure 11–4. SPI Core in a Multi-Slave Environment



Avalon Interface

The SPI core's Avalon interface consists of a single Avalon slave port. In addition to fundamental slave read and write transfers, the SPI core supports Avalon streaming read and write transfers.

Instantiating the SPI Core in SOPC Builder

The hardware feature set is configured via the SPI core's SOPC Builder configuration wizard. 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 Rate**; and **Specify Delay**.

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 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:

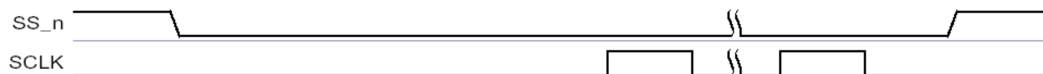
<*Avalon system clock frequency*> / [2, 4, 6, 8, ...]

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 turned on, the designer must also specify the delay time in units of ns, us or ms. An example is shown in Figure 11–5.

Figure 11–5. Time Delay Between Asserting ss_n & 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:

p = ½ * <*period of sclk*>

actual delay = ceiling(<*desired delay*> / p) * 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 11–6 through 11–9 demonstrate the behavior of signals in all possible cases of clock polarity and clock phase.

Figure 11–6. Clock Polarity = 0, Clock Phase = 0

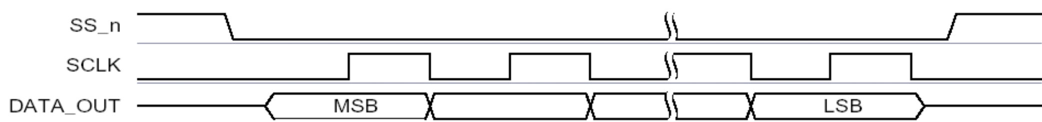


Figure 11–7. Clock Polarity = 0, Clock Phase = 1

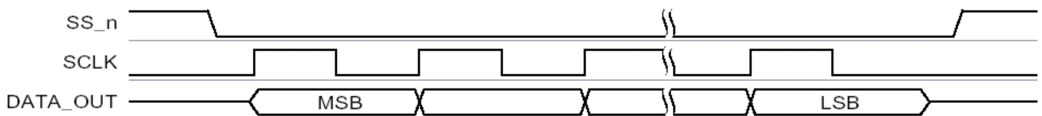


Figure 11–8. Clock Polarity = 1, Clock Phase = 0

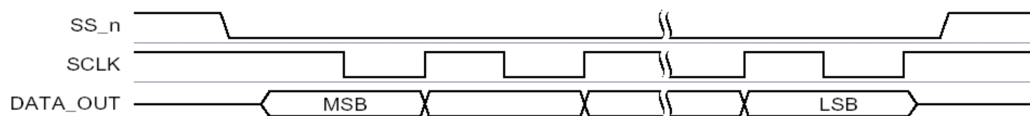
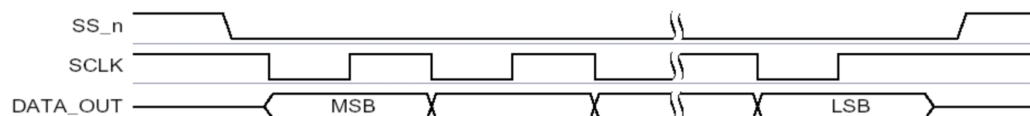


Figure 11–9. Clock Polarity = 1, Clock Phase = 1



Device & 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) Deasserts 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 on these routines, refer to the SPI documentation that accompanied the first-generation Nios processor. For details on 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 master peripheral controls and communicates with the SPI core via the six 16-bit registers, shown in Table 11–3. The table assumes an *n*-bit data width for `rxdata` and `txdata`.

Table 11–3. 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 11–3:
(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 (i.e., 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 (i.e., 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 11–14.

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 11–4 describes the individual bits of the `status` register.

Table 11–4. 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 11–5.

Table 11–5. 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 (i.e., 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.

12. EPCS Device Controller Core with Avalon Interface

NII51012-1.1

Core Overview

The EPCS device controller core with Avalon™ interface ("the EPCS controller") 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 on the EPCS serial configuration device family, see the *Serial Configuration Devices (EPCS1 & EPCS4) Data Sheet*. For details on using the Nios II HAL API to read and write flash memory, see the *Nios II Software Developer's Handbook*. For details on managing and programming the EPCS memory contents, see the *Nios II Flash Programmer User Guide*.

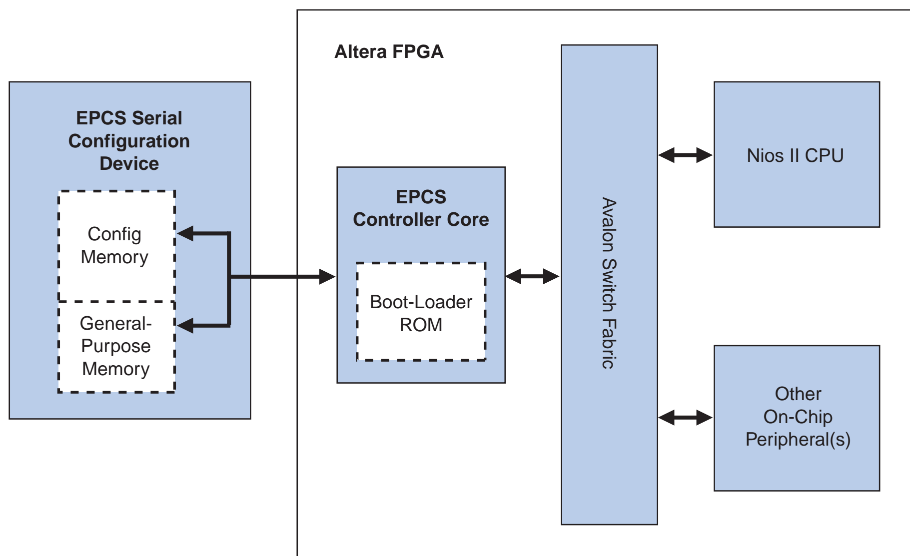
For Nios II processor users, the EPCS controller core supersedes the Active Serial Memory Interface (ASMI) device. New designs should use the EPCS controller instead of the ASMI core.

Functional Description

Figure 12–1 shows a block diagram of the EPCS controller in a typical system configuration. As shown in Figure 12–1, 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 12–1. 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 a 1 Kbyte on-chip memory for storing a boot-loader program. The Nios II processor can be configured to boot from 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. See 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. Therefore, 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 (i.e. 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 automatically routed 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.

Avalon Slave Interface & Registers

The EPCS controller core has a single Avalon slave interface that provides access to both boot-loader code and registers that control the core. As shown in Table 12–1 on page 12–4, the first 256 words are dedicated to the boot-loader code, and the next 7 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.

Table 12–1. 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 12–1:

(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 & 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 specify the core features. There is only one available option in the configuration wizard.

- **Reference Designator**—This setting is a drop-down menu that allows you to select a reference designator on the current SOPC Builder target board component, which associates the current EPCS controller to the reference designator for an EPCS device on the board. If no matching reference designator is found for the target board (i.e., the board component does not declare an EPCS device),

then an EPCS controller cannot be added to the system. The reference designator is used by the Nios II IDE flash programmer. For details see the *Nios II Flash Programmer User Guide*.

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 anything about 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 on managing and programming the EPCS device contents, see 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.

13. Common Flash Interface Controller Core with Avalon Interface

NII51013-1.2

Core Overview

The common flash interface controller core with Avalon™ interface ("the 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 on how to read and write flash using the HAL API, refer to the *Nios II Software Developer's Handbook*.

Nios II development tools provide 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 on 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**.

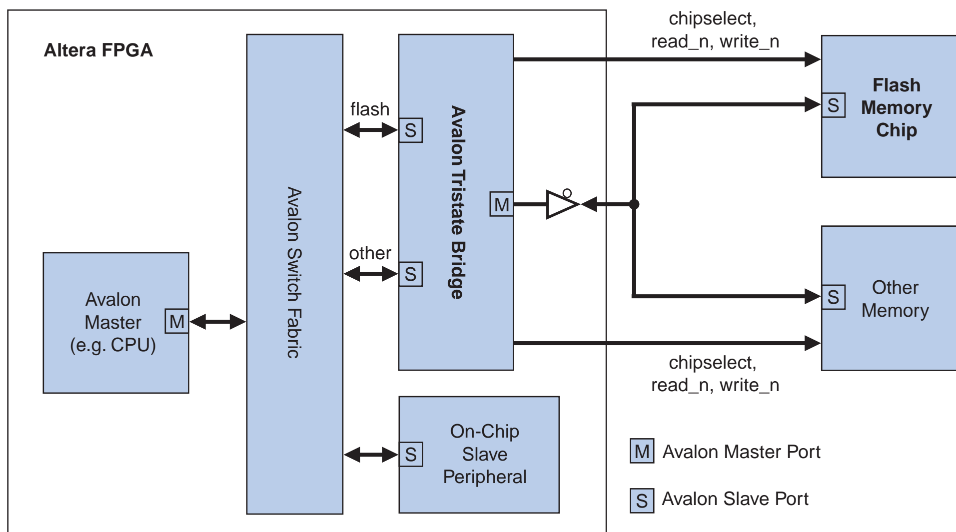
The common flash interface controller core supersedes previous Altera flash cores distributed with SOPC Builder or Nios development kits. All flash chips associated with these previous cores comply with the CFI specification, and therefore are supported by the CFI controller.

Functional Description

Figure 13–1 shows a block diagram of the CFI controller in a typical system configuration. As shown in Figure 13–1, the Avalon interface for flash devices is connected through an Avalon tristate bridge. The Avalon 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 tristate slave port configured with waitstates, setup, and hold time appropriate for the target flash chip. This slave port is capable of Avalon tristate slave read and write transfers.

Figure 13–1. An SOPC Builder System Integrating a CFI controller



Avalon master ports can perform read transfers directly from the CFI controller's Avalon port. See "Software Programming Model" on page 13–4 for more detail on writing/erasing flash memory.

Device & Tools Support

The CFI controller supports the Stratix®, Stratix II, Cyclone™, and Cyclone II 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 CFI controller's SOPC Builder configuration wizard to specify the core features. The following sections describe the available options in the configuration wizard.

Attributes Tab

The options on this tab 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 master ports of different data widths. See the *Avalon Interface Specification Reference Manual* for details about dynamic bus sizing.

Board Info

The **Board Info** setting is used by the flash programmer utility provided in Nios II development kits. This setting maps a CFI controller to a known chip in a target system board component for the SOPC Builder system.

The **Reference Designator (chip label)** setting is a drop-down menu that maps the current flash component to a reference designator on the target board. This drop-down menu is only enabled if there are multiple flash chips on the target board. If all flash chips on the board are represented by other instances of the CFI controller, SOPC Builder displays an error.

For details, see the *Nios II Flash Programmer User Guide*.

Timing Tab

The options on this tab 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 interface, see the *Avalon Interface Specification Reference Manual*.

Software Programming Model

This section describes the software programming model for the CFI controller. In general, any Avalon 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 development kit 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.

14. System ID Core with Avalon Interface

NII51014-1.1

Core Overview

The system ID core 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.

Functional Description

The system ID core provides a read-only Avalon™ slave interface. There are two registers, as shown in Table 14–1.

Table 14–1. 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 14–1:
(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 and/or different configuration options 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 & 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-settable 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 System ID configuration wizard. Hovering 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. 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.

Software Files

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.

15. Character LCD (Optrex 16207) Controller with Avalon Interface

NII51019-1.0

Core Overview

The Character LCD (Optrex 16207) Controller 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) 16x2-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.

Nios II development kits include an Optrex LCD module and provide several ready-made example designs that display text on the Optrex 16207 via the LCD controller. For details on the Optrex 16207 LCD module, see the manufacturer's *Dot Matrix Character LCD Module User's Manual* available at **http://www.optrex.com**.

Functional Description

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

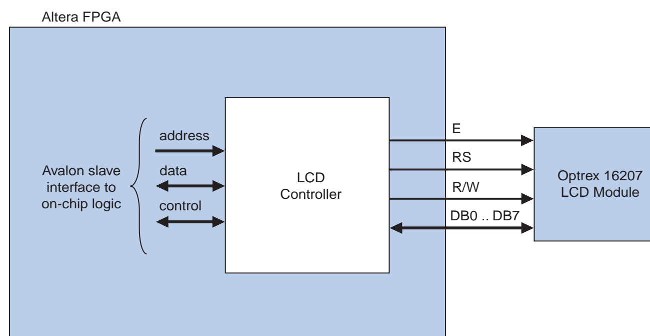
1. Eleven signals that connect to pins on the Optrex 16207 LCD panel – These signals are defined in the Optrex 16207 data sheet.

    - `E` – Enable (output)
    - `RS` – Register Select (output)
    - `R/W` – Read or Write (output)
    - `DB0` through `DB7` – Data Bus (bidirectional)

2. An Avalon 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 on the register usage. For further details, see "Software Programming Model" on page 15–2.

Figure 15–1 shows a block diagram of the LCD controller core.

Figure 15–1. LCD Controller Block Diagram



Device & 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 (16x2, 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 was 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 16x2 screen. Characters written to the LCD controller are stored to an 80-column x 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 which can be used to control the cursor position, and clear the display as shown in Table 15–1.

Table 15–1. 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 on 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.

16. Mutex Core with Avalon Interface

NII51020- 1.0

Core Overview

Multiprocessor environments can use the mutex core with Avalon™ interface (the mutex core) 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.

Functional Description

The mutex core has a simple Avalon slave interface that provides access to two memory-mapped, 32-bit registers. Table 16–1 shows the registers.

Table 16–1. 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 master peripheral) can read the `mutex` register to determine its current state.

- The `mutex` register is writeable 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 & 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 mutex core's SOPC Builder configuration wizard to specify the core's hardware features. The configuration wizard provides the following settings:

- **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 16–2.

Table 16–2. 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 16–5.

The following code demonstrates opening a mutex device handle and locking a mutex:

Example: 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.