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

This section includes the following chapters:

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

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

Core Overview

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

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

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

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

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

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

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

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

Avalon-MM Interface

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

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

For details about Avalon-MM transfer types, refer to the Avalon Interface Specifications.
Off-Chip SDRAM Interface

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

Signal Timing and Electrical Characteristics

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

Synchronizing Clock and Data Signals

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

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

For more information about instantiating a PLL in your SOPC Builder system, refer to PLL Core chapter in volume 5 of the Quartus II Handbook. The Nios® II development tools provide example hardware designs that use the SDRAM controller core in conjunction with a PLL, which you can use as a reference for your custom designs. The Nios II development tools are available free for download from www.altera.com.

Clock Enable (CKE) Not Supported

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

Sharing Pins with Other Avalon-MM Tri-State Devices

If an Avalon-MM tri-state bridge is present in the SOPC Builder system, the SDRAM controller core can share pins with the existing tri-state bridge. In this case, the core’s addr, dq (data) and dqm (byte-enable) pins are shared with other devices connected to the Avalon-MM tri-state bridge. This feature conserves I/O pins, which is valuable in systems that have multiple external memory chips (for example, flash, SRAM, and SDRAM), but too few pins to dedicate to the SDRAM chip. See “Performance Considerations” for details about how pin sharing affects performance.
The SDRAM addresses must connect all address bits regardless of the size of the word so that the low-order address bits on the tri-state bridge align with the low-order address bits on the memory device. The Avalon-MM tristate address signal always presents a byte address. It is not possible to drop A0 of the tri-state bridge for memories when the smallest access size is 16 bits or A0-A1 of the tri-state bridge when the smallest access size is 32 bits.

Board Layout and Pinout Considerations

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

Performance Considerations

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

Open Row Management

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

Sharing Data and Address Pins

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

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

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

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

- The controller cannot permanently block access to other devices sharing the tri-state bridge.
The controller is guaranteed not to violate the SDRAM’s row open time limit.

**Hardware Design and Target Device**

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

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

**Device Support**

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

**Instantiating the Core in SOPC Builder**

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

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

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

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

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

Table 1–1. Memory Profile Page Settings

<table>
<thead>
<tr>
<th>Settings</th>
<th>Allowed Values</th>
<th>Default Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>Architecture</td>
<td>Chip Selects</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></td>
<td>Banks</td>
<td>2, 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>Address Width</td>
<td>Row</td>
<td>11, 12, 13, 14</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Column</td>
<td>&gt;= 8, and less than Row value</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Share pins via tri-state bridge dq/dqm/addr I/O pins</td>
<td>On, Off</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>Include a functional memory model in the system testbench</td>
<td>On, Off</td>
<td>On</td>
</tr>
</tbody>
</table>

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

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

Table 1–2. Timing Page Settings

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

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

Hardware Simulation Considerations

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

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

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

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

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

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

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

SDRAM Memory Model

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

Using the Generic Memory Model

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

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

Using the SDRAM Manufacturer’s Memory Model

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

Example Configurations

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

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

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

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

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

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

Clock, PLL and Timing Considerations

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

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

For details about the PLL circuitry in your target device, refer to the appropriate device family handbook. For details about configuring the PLLs in Altera devices, refer to the ALTPPLL Megafunction User Guide.
Factors Affecting SDRAM Timing

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

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

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

Symptoms of an Untuned PLL

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

Estimating the Valid Signal Window

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

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

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

- Signal skew due to delays on the printed circuit board — These calculations assume zero skew.
- Delay from the PLL clock output nodes to destinations — These calculations assume that the delay from the PLL SDRAM-clock output-node to the pin is the same as the delay from the PLL controller-clock output-node to the clock inputs in the SDRAM controller. If these clock delays are significantly different, you must account for this phase shift in your window calculations.
Figure 1–5 shows how to calculate the maximum length of time that the SDRAM clock can lag the controller clock, and Figure 1–6 shows how to calculate the maximum lead. Lag is a negative time shift, relative to the controller clock, and lead is a positive time shift. The SDRAM clock can lag the controller clock by the lesser of the maximum lag for a read cycle or that for a write cycle. In other words, 
\[ \text{Maximum Lag} = \min(\text{Read Lag}, \text{Write Lag}) \]. Similarly, the SDRAM clock can lead by the lesser of the maximum lead for a read cycle or for a write cycle. In other words, 
\[ \text{Maximum Lead} = \min(\text{Read Lead}, \text{Write Lead}) \].

**Figure 1–5. Calculating the Maximum SDRAM Clock Lag**
Example Calculation

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

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

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Value (ns) in -7 Speed Grade</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access time from CLK (pos. edge)</td>
<td>CL = 3</td>
<td>t_{AC(3)}</td>
<td>—</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>CL = 2</td>
<td>t_{AC(2)}</td>
<td>—</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>CL = 1</td>
<td>t_{AC(1)}</td>
<td>—</td>
<td>17</td>
</tr>
<tr>
<td>Address hold time</td>
<td>t_{AH}</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Address setup time</td>
<td>t_{AS}</td>
<td>2</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CLK high-level width</td>
<td>t_{CH}</td>
<td>2.75</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CLK low-level width</td>
<td>t_{CL}</td>
<td>2.75</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Table 1–4 shows the relevant timing information, obtained from the Timing Analyzer section of the Quartus II Compilation Report. The values in the table are the maximum or minimum values among all device pins related to the SDRAM. The variance in timing between the SDRAM pins on the device is small (less than 100 ps) because the registers for these signals are placed in the I/O cell.

Table 1–4. FPGA I/O Timing Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Value (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period</td>
<td>$t_{CLK}$</td>
<td>20</td>
</tr>
<tr>
<td>Minimum clock-to-output time</td>
<td>$t_{CL, min}$</td>
<td>2.399</td>
</tr>
<tr>
<td>Maximum clock-to-output time</td>
<td>$t_{CL, max}$</td>
<td>2.477</td>
</tr>
<tr>
<td>Maximum hold time after clock</td>
<td>$t_{H, max}$</td>
<td>−5.607</td>
</tr>
<tr>
<td>Maximum setup time before clock</td>
<td>$t_{SU, max}$</td>
<td>5.936</td>
</tr>
</tbody>
</table>

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

The timing values found in the compilation report can change, depending on fitting, pin location, and other Quartus II logic settings. When you recompile the design in the Quartus II software, verify that the I/O timing has not changed significantly.
The following examples illustrate the calculations from Figure 1–5 and Figure 1–6 using the values from Table 1–3 and Table 1–4.

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

\[
\text{Read Lag} = t_{OH}(SDRAM) - t_{H_{\text{MAX}}}(FPGA)
\]
\[
= 2.5 \text{ ns} - (-5.607 \text{ ns}) = 8.107 \text{ ns}
\]

or

\[
\text{Write Lag} = t_{CLK} - t_{CO_{\text{MAX}}}(FPGA) - t_{DS}(SDRAM)
\]
\[
= 20 \text{ ns} - 2.477 \text{ ns} - 2 \text{ ns} = 15.523 \text{ ns}
\]

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

\[
\text{Read Lead} = t_{CO_{\text{MIN}}}(FPGA) - t_{DH}(SDRAM)
\]
\[
= 2.399 \text{ ns} - 1.0 \text{ ns} = 1.399 \text{ ns}
\]

or

\[
\text{Write Lead} = t_{CLK} - t_{HZO}(SDRAM) - t_{SU_{\text{MAX}}}(FPGA)
\]
\[
= 20 \text{ ns} - 5.5 \text{ ns} - 5.936 \text{ ns} = 8.564 \text{ ns}
\]

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

**Referenced Documents**

This chapter references the following documents:

- ALTPLL Megafunction User Guide
- AN 351: Simulating Nios II Processor Designs
- Avalon Interface Specifications
- PLL Core chapter in volume 5 of the Quartus II Handbook
Document Revision History

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

Table 1–5. Document Revision History

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>No change from previous release.</td>
<td></td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>No change from previous release.</td>
<td></td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td></td>
</tr>
<tr>
<td>May 2008 v8.0.0.</td>
<td>No change from previous release.</td>
<td></td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.
Core Overview

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

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

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

This chapter contains the following sections:

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

Functional Description

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

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

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

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

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

### Instantiating the Core in SOPC Builder

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

### Required Connections

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

<table>
<thead>
<tr>
<th>CompactFlash Interface Signal Name</th>
<th>Pin Type</th>
<th>CompactFlash Pin Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr [0]</td>
<td>Output</td>
<td>20</td>
</tr>
<tr>
<td>addr [1]</td>
<td>Output</td>
<td>19</td>
</tr>
<tr>
<td>addr [2]</td>
<td>Output</td>
<td>18</td>
</tr>
<tr>
<td>addr [3]</td>
<td>Output</td>
<td>17</td>
</tr>
<tr>
<td>addr [4]</td>
<td>Output</td>
<td>16</td>
</tr>
<tr>
<td>addr [5]</td>
<td>Output</td>
<td>15</td>
</tr>
<tr>
<td>addr [6]</td>
<td>Output</td>
<td>14</td>
</tr>
<tr>
<td>addr [7]</td>
<td>Output</td>
<td>12</td>
</tr>
<tr>
<td>addr [8]</td>
<td>Output</td>
<td>11</td>
</tr>
<tr>
<td>addr [9]</td>
<td>Output</td>
<td>10</td>
</tr>
<tr>
<td>addr [10]</td>
<td>Output</td>
<td>8</td>
</tr>
<tr>
<td>atasel_n</td>
<td>Output</td>
<td>9</td>
</tr>
<tr>
<td>cs_n [0]</td>
<td>Output</td>
<td>7</td>
</tr>
<tr>
<td>cs_n [1]</td>
<td>Output</td>
<td>32</td>
</tr>
<tr>
<td>data [0]</td>
<td>Input/Output</td>
<td>21</td>
</tr>
<tr>
<td>data [1]</td>
<td>Input/Output</td>
<td>22</td>
</tr>
</tbody>
</table>
Device Support

The CompactFlash interface core supports all Altera® device families.

Software Programming Model

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

HAL System Library Support

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

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

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

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

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

Register Maps

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

Ide Registers

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

Table 2–2. Ide Register Map

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Names</th>
<th>Read Operation</th>
<th>Write Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RD Data</td>
<td>WR Data</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Error</td>
<td>Features</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Sector Count</td>
<td>Sector Count</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Sector No</td>
<td>Sector No</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cylinder Low</td>
<td>Cylinder Low</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cylinder High</td>
<td>Cylinder High</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Select Card/Head</td>
<td>Select Card/Head</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Status</td>
<td>Command</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Alt Status</td>
<td>Device Control</td>
<td></td>
</tr>
</tbody>
</table>

Ctl Registers

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

Table 2–3. Ctl Register Map

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cfctl</td>
<td>31:4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>idectl</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
Cfctl Register

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

Table 2–4. cfctl Register Bits

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

Idectl Register

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

Table 2–5. idectl Register

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Bit Name</th>
<th>Read/Write</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IIDE</td>
<td>RW</td>
<td>IDE Interrupt Enable. When this bit is set to 1, the CompactFlash core generates an interrupt following an interrupt generated by the CompactFlash device. Setting this bit to 0 disables the IDE interrupt.</td>
</tr>
</tbody>
</table>

Document Revision History

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

Table 2–6. Document Revision History

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td>—</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>Added the mode supported by the CompactFlash core.</td>
<td>—</td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.
Core Overview

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

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

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

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

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

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

This chapter contains the following sections:

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

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

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

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

Device and Tools Support

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

Instantiating the Core in SOPC Builder

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

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

Presets Settings

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

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

Size Settings

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

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

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

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

Timing Page

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

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

The settings available on the Timing page are:

- Setup—After asserting chipselect, the time required before asserting the read or write signals. You can determine the value of this parameter by using the following formula:
  
  \[
  \text{Setup} = t_{CE} (\text{chip enable to output delay}) - t_{OE} (\text{output enable to output delay})
  \]

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

  The sum of Setup, Wait, and board delay must be greater than tACC, where:

  - Board delay is determined by the TCO on the device address pins, TSU on the device data pins and propagation delay on the board traces in both directions.
  - tACC is the address to output delay.
Hold—After deasserting the write signal, the time required before deasserting the chipselect signal.

Units—The timing units used for the Setup, Wait, and Hold values. Possible values include ns, µs, ms, and clock cycles.

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

**Software Programming Model**

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

**HAL System Library Support**

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

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

**Limitations**

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

**Software Files**

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

- altera_avalon_cfi_flash.h, altera_avalon_cfi_flash.c—The header and source code for the functions and variables required to integrate the driver into the HAL system library.

- altera_avalon_cfi_flashfuncs.h, altera_avalon_cfi_flash_table.c—The header and source code for functions concerned with accessing the CFI table.

- altera_avalon_cfi_flash_amd_funcs.h, altera_avalon_cfi_flash_amd.c—The header and source code for programming AMD CFI-compliant flash chips.

- altera_avalon_cfi_flash_intel_funcs.h, altera_avalon_cfi_flash_intel.c—The header and source code for programming Intel CFI-compliant flash chips.
Referenced Documents

This chapter references the following documents:

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

Document Revision History

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

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

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.
4. EPCS Device Controller Core

Core Overview

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

Using the EPCS device controller core, Nios II systems can:

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

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

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

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

This chapter contains the following sections:

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

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

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

![Figure 4–1. Nios II System Integrating an EPCS Device Controller Core](image)

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

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

---

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

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

**Figure 4–1. Nios II System Integrating an EPCS Device Controller Core**

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

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

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

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

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

You have the flexibility to connect the output pins of Cyclone III devices, which are exported to the top-level design, to any EPCS devices. Perform the following tasks in the Quartus® II software to make the necessary pin assignments:

- On the Dual-purpose pins page (Assignments > Devices > Device and Pin Options), ensure that the following pins are assigned to the respective values:
  - Data[0] = Use as regular I/O
  - Data[1] = Use as regular I/O
  - DCLK = Use as regular I/O
  - FLASH_nCE/nCS0 = Use as regular I/O

- Using the Pin Planner (Assignments > Pins), ensure that the following pins are assigned to the respective configuration functions on the device:
  - data0_to_the_epcs_controller = DATA0
  - sdo_from_the_epcs_controller = DATA1,ASDO
  - dclk_from_epcs_controller = DCLK
  - sce_from_the_epcs_controller = FLASH_nCE

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

**Avalon-MM Slave Interface and Registers**

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

The EPCS device controller core includes an interrupt signal that can be used to interrupt the CPU when a transfer has completed.
The EPCS device controller core supports all Altera device families except the Hardcopy® series. The core must be connected to a Nios II processor. The core provides drivers for HAL-based Nios II systems, and the precompiled boot loader code compatible with the Nios II processor.

### Instantiating the Core in SOPC Builder

You can add the EPCS device controller core from the System Contents tab in SOPC Builder. There are no user-configurable settings for this component.

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

### Software Programming Model

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

### HAL System Library Support

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

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

### Table 4–1. EPCS Device Controller Core Register Map

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

Note to Table 4–1:

1. Altera does not publish the usage of the control and data registers. To access the EPCS device, you must use the HAL drivers provided by Altera.
#define ALT_USE_EPCS_FLASH
#include <altera_avalon_epcs_flash_controller.h>

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

### Software Files

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

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

### Referenced Documents

This chapter references the following documents:

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

### Document Revision History

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

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>■ Revised descriptions of register fields and bits.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Updated the section on HAL System Library Support.</td>
<td></td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>Updated the boot ROM memory offset for other device families in Table 4–1.</td>
<td></td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td></td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>■ Updated the boot rom size.</td>
<td>Updates made to comply with the Quartus II software version 8.0 release.</td>
</tr>
<tr>
<td></td>
<td>■ Added additional steps to perform to connect output pins in Cyclone III devices.</td>
<td></td>
</tr>
</tbody>
</table>
For previous versions of the *Quartus II Handbook*, refer to the *Quartus II Handbook Archive*. 
5. JTAG UART Core

Core Overview

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

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

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

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

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

Figure 5–1. JTAG UART Core Block Diagram

Avalon Slave Interface and Registers

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

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

Read and Write FIFOs

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

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

Host-Target Connection

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

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

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

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

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

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

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

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

Instantiating the Core in SOPC Builder

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

Configuration Page

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

Write FIFO Settings

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

- **Depth**—The write FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.

- **IRQ Threshold**—The write IRQ threshold governs how the core asserts its IRQ in response to the FIFO emptying. As the JTAG circuitry empties data from the write FIFO, the core asserts its IRQ when the number of characters remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by writing more data and preventing the write FIFO from emptying completely. A value of 8 is typically optimal. See “Interrupt Behavior” on page 5–11 for further details.

- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of on-chip logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 logic elements (LEs), so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.
Read FIFO Settings
The read FIFO buffers data flowing from the host to the Avalon interface. Settings are available to control the depth of the FIFO and the generation of interrupts.

- **Depth**—The read FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.

- **IRQ Threshold**—The IRQ threshold governs how the core asserts its IRQ in response to the FIFO filling up. As the JTAG circuitry fills up the read FIFO, the core asserts its IRQ when the amount of space remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by reading data and preventing the read FIFO from filling up completely. A value of 8 is typically optimal. See “Interrupt Behavior” on page 5–11 for further details.

- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 LEs, so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.

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

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

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

- **Do not generate ModelSim aliases for interactive windows**—This option does not create any ModelSim macros for character I/O.

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

Hardware Simulation Considerations

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

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

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

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

Software Programming Model

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

HAL System Library Support

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

If your program uses the Altera-provided HAL device driver to access the JTAG UART hardware, accessing the device registers directly will interfere with the correct behavior of the driver.
For Nios II processor users, the HAL system library API provides complete access to the JTAG UART core’s features. Nios II programs treat the JTAG UART core as a character mode device, and send and receive data using the ANSI C standard library functions, such as `getchar()` and `printf()`.

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

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

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

```c
#include <stdio.h>

int main ()
{
    printf("Hello world.\n");
    return 0;
}
```

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

```c
/* 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
                clearerr(fp);// If so, clear it.
        }

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

    return 0;
}
```
In this example, the `ferror(fp)` is used to check if an error occurred on the JTAG UART connection, such as a disconnected JTAG connection. In this case, the driver detects that the JTAG connection is disconnected, reports an error (EIO), and discards data for subsequent transactions. If this error ever occurs, the C library latches the value until you explicitly clear it with the `clearerr()` function.

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

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

**Driver Options: Fast vs. Small Implementations**

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

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

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

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

**ioctl() Operations**

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

<table>
<thead>
<tr>
<th>Request</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIOCSTIMEOUT</td>
<td>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.</td>
</tr>
<tr>
<td>TIOCGCONNECTED</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

Software Files

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

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

Accessing the JTAG UART Core via a Host PC

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

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

Register Map

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

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

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

| Offset | Register Name | R/W | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 |
| 0      | data          | RW  |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | RAVAIL | RVALID | Reserved | DATA |
| 1      | control       | RW  |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | WSPACE | Reserved | AC | WI | RI | Reserved | WE | RE |

Note to Table 5–2:
(1) Reserved fields—Read values are undefined. Write zero.

Table 5–3. data Register Bits

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

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

Control Register

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

Table 5–4. Control Register Bits

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

A read from the control register returns the status of the read and write FIFOs. Writes to the register can be used to enable/disable interrupts, or clear the AC bit.
The RE and WE bits enable interrupts for the read and write FIFOs, respectively. The WI and RI bits indicate the status of the interrupt sources, qualified by the values of the interrupt enable bits (WE and RE). Embedded software can examine RI and WI to determine the condition that generated the IRQ. See “Interrupt Behavior” on page 5–11 for further details.

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

**Interrupt Behavior**

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

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

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

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

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

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

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

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

Document Revision History

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

Table 5–5. Document Revision History

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td>—</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
</tbody>
</table>

For previous versions of the *Quartus II Handbook*, refer to the *Quartus II Handbook Archive*. 
6. UART Core

Core Overview

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

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

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

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

Functional Description

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

Figure 6–1. Block Diagram of the UART Core in a Typical System
The core has two user-visible parts:

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

**Avalon-MM Slave Interface and Registers**

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

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

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

For more information, refer to the Timer Core chapter in volume 5 of the Quartus II Handbook. For details about the Avalon-MM interface, refer to the Avalon Interface Specifications.

**RS-232 Interface**

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

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

**Transmitter Logic**

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

These two registers provide double buffering. A master peripheral can write a new value into the txdata register while the previously written character is being shifted out. The master peripheral can monitor the transmitter’s status by reading the status register’s transmitter ready (TRDY), transmitter shift register empty (tmt), and transmitter overrun error (TOE) bits.
The transmitter logic automatically inserts the correct number of start, stop, and parity bits in the serial TXD data stream as required by the RS-232 specification.

**Receiver Logic**

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

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

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

**Baud Rate Generation**

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

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

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

**Device Support**

The UART core supports all Altera® device families.

**Instantiating the Core in SOPC Builder**

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

Use the MegaWizard™ interface for the UART core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.
Chapter 6: UART Core

Instantiating the Core in SOPC Builder

Configuration Settings

This section describes the configuration settings.

Baud Rate Options

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

- **Fixed rate**—The baud rate is fixed at system generation time and cannot be changed via the Avalon-MM slave port.

- **Variable rate**—The baud rate can vary, based on a clock divisor value held in the divisor register. A master peripheral changes the baud rate by writing new values to the divisor register.

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

Baud Rate (bps) Setting

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

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

**Equation 6–1.**

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

**Equation 6–2.**

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

Baud Rate Can Be Changed By Software Setting

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

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

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

Table 6–1. Data Bits Settings

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

Synchronizer Stages

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

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

Flow Control

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

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

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

**Streaming Data (DMA) Control**

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

**Include End-of-Packet Register**

When this setting is on, the UART core includes:

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

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

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

**Simulation Settings**

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

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

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

Prepare Interactive Windows

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

- **Create ModelSim alias to open streaming output window** to create a ModelSim macro that opens a window to display all output from the TXD port.

- **Create ModelSim alias to open interactive stimulus window** to create a ModelSim macro that opens a window to accept stimulus for the RXD port. The window sends any characters typed in the window to the RXD port.

Simulated Transmitter Baud Rate

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

- **Accelerated (use divisor = 2)**—TXD emits one bit per 2 clock cycles in simulation.

- **Actual (use true baud divisor)**—TXD transmits at the actual baud rate, as determined by the divisor register.

Simulation Considerations

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

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

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

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

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

HAL System Library Support

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

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

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

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

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

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

```c
#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.
Driver Options: Fast Versus Small Implementations

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

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

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

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

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

Example 6–2. Example: Sending and Receiving Characters

```c
/* 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')
        {
            prompt = getc(fp); // Get a character from the UART.
            if (prompt == 't')
            {
                fprintf(fp, "Closing the UART file.\n");
                fwrite(msg, strlen(msg), 1, fp);
            }
        }
        fprintf(fp, "Closing the UART file.\n");
        fclose(fp);
    }
    return 0;
}
```

For more information about the HAL system library, refer to the *Nios II Software Developer’s Handbook.*
If the CTS/RTS flow control signals are enabled in hardware, the fast driver automatically uses them. The small driver always ignores them.

**ioctl() Operations**

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

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

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

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

<table>
<thead>
<tr>
<th>Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIOCMGET</td>
<td>Returns the current configuration of the device by filling in the contents of the input termios structure. (1) A pointer to this structure is supplied as the value of the parameter <code>opt</code>.</td>
</tr>
<tr>
<td>TIOCMSET</td>
<td>Sets the configuration of the device according to the values contained in the input termios structure. (1) A pointer to this structure is supplied as the value of the parameter <code>arg</code>.</td>
</tr>
</tbody>
</table>

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

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

**Limitations**

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

**Software Files**

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

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

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

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

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

Table 6–4. UART Core Register Map

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>R/W</th>
<th>Description/Register Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rxdata</td>
<td>RO</td>
<td>(1) (1) Receive Data</td>
</tr>
<tr>
<td>1</td>
<td>txdata</td>
<td>WO</td>
<td>(1) (1) Transmit Data</td>
</tr>
<tr>
<td>2</td>
<td>status (2)</td>
<td>RW</td>
<td>eop cts dcts (1) e rrdy trdy tmt toe roe brk fe pe</td>
</tr>
<tr>
<td>3</td>
<td>control</td>
<td>RW</td>
<td>leop rts idcts trbk ie irrdy itrdy itmt itoe iroe ibrk ifeipe</td>
</tr>
<tr>
<td>4</td>
<td>divisor (3)</td>
<td>RW</td>
<td>Baud Rate Divisor</td>
</tr>
<tr>
<td>5</td>
<td>endof-packet (3)</td>
<td>RW</td>
<td>(1) (1) End-of-Packet Value</td>
</tr>
</tbody>
</table>

Notes to Table 6–4:

(1) These bits may or may not exist, depending on the Data Width hardware option. If they do not exist, they read zero, and writing has no effect.
(2) Writing zero to the status register clears the dcts, e, toe, roe, brk, fe, and pe bits.
(3) This register may or may not exist, depending on hardware configuration options. If it does not exist, reading returns an undefined value and writing has no effect.

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

rxdata Register

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

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

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

status Register

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

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>PE</td>
<td>RC</td>
<td>Parity error. A parity error occurs when the received parity bit has an unexpected (incorrect) logic level. The PE bit is set to 1 when the core receives a character with an incorrect parity bit. The PE bit stays set to 1 until it is explicitly cleared by a write to the status register. When the PE bit is set, reading from the rxdata register produces an undefined value. If the Parity hardware option is not enabled, no parity checking is performed and the PE bit always reads 0. Refer to “Data Bits, Stop Bits, Parity” on page 6–5.</td>
</tr>
<tr>
<td>1</td>
<td>PE</td>
<td>RC</td>
<td>Framing error. A framing error occurs when the receiver fails to detect a correct stop bit. The PE bit is set to 1 when the core receives a character with an incorrect stop 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.</td>
</tr>
<tr>
<td>2</td>
<td>BRK</td>
<td>RC</td>
<td>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.</td>
</tr>
<tr>
<td>3</td>
<td>ROE</td>
<td>RC</td>
<td>Receive overrun error. A receive-overrun error occurs when a newly received character is transferred into the rxdata holding register before the previous character is read (in other words, while the RRDY bit is 1). In this case, the ROE bit is set to 1, and the previous contents of rxdata are overwritten with the new character. The ROE bit stays set to 1 until it is explicitly cleared by a write to the status register.</td>
</tr>
</tbody>
</table>
Table 6–5. status Register Bits (Part 2 of 2)

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

Note to Table 6–5:
(1) This bit is optional and may not exist in hardware.
control Register

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

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

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

### Table 6–6. control Register Bits

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

**Note to Table 6–6:**

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

divisor Register (Optional)

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

\[
\text{Baud Rate} = \frac{\text{(Clock frequency)}}{\text{(divisor + 1)}}
\]

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

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

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

Interrupt Behavior

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

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

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

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

Referenced Documents

This chapter references the following documents:

- AN 350: Upgrading Nios Processor Systems to the Nios II Processor
- AN 351: Simulating Nios II Embedded Processor Designs
- Avalon Interface Specifications
- Nios II Software Developer’s Handbook
- Timer Core chapter in volume 5 of the Quartus II Handbook
- AN 42: Metastability in Altera Devices
- Area and Timing Optimization chapter in volume 2 of the Quartus II Handbook
Table 6–7 shows the revision history for this chapter.

### Table 6–7. Document Revision History

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>Added description of a new parameter, <em>Synchronizer stages</em>.</td>
<td>—</td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td>—</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
</tbody>
</table>

For previous versions of the *Quartus II Handbook*, refer to the *Quartus II Handbook Archive*. 
Core Overview

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

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

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

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

Functional Description

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

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

The SPI core has the following user-visible features:

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

The registers provide an interface to the SPI core and are visible via the Avalon-MM slave port. The sclk, ss_n, mosi, and miso ports provide the hardware interface to other SPI devices. The behavior of sclk, ss_n, mosi, and miso depends on whether the SPI core is configured as a master or slave.
Figure 7–1 shows a block diagram of the SPI core in master mode.

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

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

For more details, refer to the Interval Timer Core chapter in volume 5 of the Quartus II Handbook.

**Example Configurations**

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

**Figure 7–2.** SPI Core Configured as a Slave

In Figure 7–1, the SPI core provides a master interface driving multiple off-chip slave devices. Each slave device in Figure 7–1 must tristate its miso output whenever its select signal is not asserted.
The $ss_n$ signal is active-low. However, any signal can be inverted inside the FPGA, allowing the slave-select signals to be either active high or active low.

**Transmitter Logic**

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

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

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

**Receiver Logic**

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

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

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

**Master and Slave Modes**

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

**Master Mode Operation**

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mosi</td>
<td>output</td>
<td>Data output to slave(s)</td>
</tr>
<tr>
<td>miso</td>
<td>input</td>
<td>Data input from slave(s)</td>
</tr>
</tbody>
</table>
In master mode, an intelligent host (for example, a microprocessor) configures the SPI core using the control and slaveselect registers, and then writes data to the txdata buffer to initiate a transaction. A master peripheral can monitor the status of the transaction by reading the status register. A master peripheral can enable interrupts to notify the host whenever new data is received (for example, a transfer has completed), or whenever the transmit buffer is ready for new data.

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

When the SPI core is configured to interface with multiple slaves, the core has one ss_n signal for each slave, 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 contention on the miso input. The number of slave devices is specified at system generation time.

### Slave Mode Operation

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mosi</td>
<td>input</td>
<td>Data input from the master</td>
</tr>
<tr>
<td>miso</td>
<td>output</td>
<td>Data output to the master</td>
</tr>
<tr>
<td>sclk</td>
<td>input</td>
<td>Synchronization clock</td>
</tr>
<tr>
<td>ss_n</td>
<td>input</td>
<td>Select signal</td>
</tr>
</tbody>
</table>

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

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

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

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

Avalon-MM Interface

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

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

Instantiating the SPI Core in SOPC Builder

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

Master/Slave Settings

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

Number of Select (\texttt{SS\_n}) Signals

This setting specifies how many slaves the SPI master connects to. The range is 1 to 32. The SPI master core presents a unique \texttt{ss\_n} signal for each slave.
SPI Clock (sclk) Rate
This setting determines the rate of the sclk signal that synchronizes data between master and slaves. The target clock rate can be specified in units of Hz, kHz or MHz. The SPI master core uses the Avalon-MM system clock and a clock divisor to generate sclk.

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

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

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

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

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

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

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

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

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

- **Width**—This setting specifies the width of rxdata, txdata, and the receive and transmit shift registers. The range is from 1 to 32.

- **Shift direction**—This setting determines the direction that data shifts (MSB first or LSB first) into and out of the shift registers.
**Timing Settings**

The timing settings affect the timing relationship between the \textit{ss\_n}, \textit{sclk}, \textit{mosi} and \textit{miso} signals. In this discussion the \textit{mosi} and \textit{miso} signals are referred to generically as \textit{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 \textit{sclk} is low. When clock polarity is set to 1, the idle state for \textit{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 \textit{sclk}, and data changes on trailing edge. When clock phase is 1, data is latched on the trailing edge of \textit{sclk}, and data changes on the leading edge.

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

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

![Figure 7–5](image)

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

![Figure 7–6](image)

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

![Figure 7–7](image)

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

![Figure 7–8](image)
Device Support

The SPI core supports all Altera® device families.

Software Programming Model

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

Hardware Access Routines

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

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

Thread-safe: No.
Available from ISR: No.
Include: <altera_avalon_spi.h>
Description: alt_avalon_spi_command() is used to perform a control sequence on the SPI bus. This routine is designed for SPI masters of 8-bit data width or less. Currently, it does not support SPI hardware with data-width greater than 8 bits. A single call to this function writes a data buffer of arbitrary length out the MOSI port, and then reads back an arbitrary amount of data from the MISO port. The function performs the following actions:

(1) Asserts the slave select output for the specified slave. The first slave select output is numbered 0, the next is 1, etc.

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

(3) Reads read_length bytes of data, storing the data into the buffer pointed to by read_data. MOSI is set to zero during the read transaction.

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

This function is not thread safe. If you want to access the SPI bus from more than one thread, 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.

Register Map

An Avalon-MM master peripheral controls and communicates with the SPI core via the six 32-bit registers, shown in Table 7–3. The table assumes an $n$-bit data width for rxdata and txdata.
Reading undefined bits returns an undefined value. Writing to undefined bits has no effect.

**rxdata Register**

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

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

**txdata Register**

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

A master peripheral should not write to the txdata register until the transmitter is ready for new data. If TRDY is 0 and a master peripheral writes new data to the txdata register, a transmit-overrun error occurs and the status register’s TOE bit is set to 1. In this case, the new data is ignored, and the content of txdata remains unchanged.
As an example, assume that the SPI core is idle (that is, the \texttt{txdata} register and transmit shift register are empty), when a CPU writes a data value into the \texttt{txdata} holding register. The \texttt{TRDY} bit is set to 0 momentarily, but after the data in \texttt{txdata} is transferred into the transmitter shift register, \texttt{TRDY} returns to 1. The CPU writes a second data value into the \texttt{txdata} register, and again the \texttt{TRDY} bit is set to 0. This time the shift register is still busy transferring the original data value, so the \texttt{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 \texttt{TRDY} bit is again set to 1.

**status Register**

The \texttt{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 \texttt{control} register, as discussed in “control Register” on page 7–12. A master peripheral can read \texttt{status} at any time without changing the value of any bits. Writing \texttt{status} does clear the \texttt{ROE}, \texttt{TOE} and \texttt{E} bits. Table 7–4 describes the individual bits of the \texttt{status} register.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 3 | ROE  | Receive-overrun error  
The ROE bit is set to 1 if new data is received while the \texttt{rxdata} register is full (that is, while the \texttt{RRDY} bit is 1). In this case, the new data overwrites the old. Writing to the \texttt{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 \texttt{txdata} register while it is still full (that is, while the \texttt{TRDY} bit is 0). In this case, the new data is ignored. Writing to the \texttt{status} register clears the TOE bit to 0. |
| 5 | TMT  | Transmitter shift-register empty  
In master mode, the TMT bit is set to 0 when a transaction is in progress and set to 1 when the shift register is empty.  
In slave mode, the TMT bit is set to 0 when the slave is selected (\texttt{SS_n} is low) or when the SPI Slave register interface is not ready to receive data. |
| 6 | TRDY | Transmitter ready  
The TRDY bit is set to 1 when the \texttt{txdata} register is empty. |
| 7 | RRDY | Receiver ready  
The RRDY bit is set to 1 when the \texttt{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 \texttt{status} register clears the E bit to 0. |
control Register

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

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

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

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

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

slaveselect Register

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

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

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

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

Referenced Documents

This chapter references the following documents:

- AN 350: Upgrading Nios Processor Systems to the Nios II Processor
- Interval Timer Core chapter in volume 5 of the Quartus II Handbook
Document Revision History

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

Table 7–6. Document Revision History

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

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.
Core Overview

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

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

This chapter contains the following sections:
- “Functional Description”
- “Device and Tools Support” on page 8–2
- “Instantiating the Core in SOPC Builder” on page 8–2
- “Software Programming Model” on page 8–2

Functional Description

The LCD controller core consists of two user-visible components:
- Eleven signals that connect to pins on the Optrex 16207 LCD panel—These signals are defined in the Optrex 16207 data sheet.
  - E—Enable (output)
  - RS—Register Select (output)
  - R/W—Read or Write (output)
  - DB0 through DB7—Data Bus (bidirectional)
- An Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to 4 registers.
Figure 8–1 shows a block diagram of the LCD controller core.

**Figure 8–1. LCD Controller Block Diagram**

![LCD Controller Block Diagram](image)

---

**Device and Tools Support**

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

**Instantiating the Core in SOPC Builder**

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

**Software Programming Model**

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

**HAL System Library Support**

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

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

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

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

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

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

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

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

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

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

Software Files

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

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

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

Interrupt Behavior

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

Referenced Documents

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

Document Revision History

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

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td>—</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
</tbody>
</table>

For previous versions of the *Quartus II Handbook*, refer to the *Quartus II Handbook Archive*. 
Core Overview

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

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

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

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

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

Functional Description

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

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

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

**Data Input and Output**

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

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

**Edge Capture**

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

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

- **Level-sensitive**—The PIO core hardware can detect a high level. A NOT gate can be inserted external to the core to provide negative sensitivity.

- **Edge-sensitive**—The core’s edge capture configuration determines which type of edge causes an IRQ

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

Example Configurations

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

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

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

**Figure 9–3.** PIO Core with Bidirectional Ports

Avalon-MM Interface

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

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

Basic Settings

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

**Width**

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

**Direction**

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

**Output Port Reset Value**

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

**Output Register**

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

**Input Options**

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

**Edge Capture Register**

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

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

Turn on **Enable bit-clearing for edge capture register** to clear individual bit in the edge capture register. To clear a given bit, write 1 to the bit in the edge capture register.

### Interrupt

**Interrupt**

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

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

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

### Simulation

**Simulation**

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

### Device Support

**Device Support**

The PIO core supports all Altera® device families.

### Software Programming Model

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

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

### Software Files

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

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

Table 9–2. Register Map for the PIO Core

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>R/W</th>
<th>(n-1)</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>data</td>
<td>R</td>
<td>( n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>W</td>
<td>( n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>direction (1)</td>
<td>R/W</td>
<td>( n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>interruptmask (1)</td>
<td>R/W</td>
<td>( n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>edgecapture (1), (2)</td>
<td>R/W</td>
<td>( n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>outset</td>
<td>W</td>
<td>( n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>outclear</td>
<td>W</td>
<td>( n )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes to Table 9–2:

1. This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.
2. Writing any value to edgecapture clears all bits to 0.

**data Register**

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

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

**direction Register**

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

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

After reset, all bits of direction are 0, so that all bidirectional I/O ports are configured as inputs. If those PIO ports are connected to device pins, the pins are held in a high-impedance state. In bi-directional mode, to change the direction of the PIO port, reprogram the direction register.
interruptmask Register
Setting a bit in the interruptmask register to 1 enables interrupts for the corresponding PIO input port. Interrupt behavior depends on the hardware configuration of the PIO core. See “Interrupt Behavior” on page 9–7.

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

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

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

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

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

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

Interrupt Behavior
The PIO core outputs a single IRQ signal that can connect to any master peripheral in the system. The master can read either the data register or the decapture 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 decapture and interruptmask registers are 1. The IRQ remains asserted until explicitly acknowledged by disabling the appropriate bit(s) in interruptmask, or by writing to decapture.

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

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


## Document Revision History

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

**Table 9–3. Document Revision History**

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

For previous versions of the *Quartus II Handbook*, refer to the *Quartus II Handbook Archive*. 
Core Overview

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

The SPI Slave to Avalon Master Bridge is an example of how this core is used. For more information on the bridge, refer to the SPI Slave/JTAG to Avalon Master Bridge Cores chapter in volume 5 of the Quartus II Handbook.

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

This chapter contains the following sections:

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

Functional Description

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

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

The serial peripheral interface is full-duplex and does not support backpressure. It supports SPI clock phase bit, CPHA = 1, and SPI clock polarity bit, CPOL = 0. Table 10–1 shows the properties of the Avalon-ST interfaces.

Table 10–1. Properties of Avalon-ST Interfaces

<table>
<thead>
<tr>
<th>Feature</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backpressure</td>
<td>Not supported.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Data width = 8 bits; Bits per symbol = 8.</td>
</tr>
<tr>
<td>Channel</td>
<td>Not supported.</td>
</tr>
<tr>
<td>Error</td>
<td>Not used.</td>
</tr>
<tr>
<td>Packet</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>

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

Operation

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

- \texttt{0x4a}—Idle character. The core drops the idle character.
- \texttt{0x4d}—Escape character. The core drops the escape character, and XORs the following byte with \texttt{0x20}.

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

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

The data shifts into and out of the core in the direction of MSB first.
Figure 10–2 shows the SPI transfer protocol.

**Figure 10–2. SPI Transfer Protocol**

Notes to Figure 10–2:
1. TL = The worst recovery time of \textit{sclk} with respect with \textit{nSS}.
2. TT = The worst hold time for \textit{MOSI} and \textit{MISO} data.
3. TI = The minimum width of a reset pulse required by Altera FPGA families.

**Timing**

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

**Limitations**

Daisy-chain configuration, where the output line \textit{MISO} of an instance of the core is connected to the input line \textit{MOSI} of another instance, is not supported.

**Instantiating the Core in SOPC Builder**

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

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

**Device Support**

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

This chapter references the following documents:

- Avalon Interface Specifications
- SPI Slave/JTAG to Avalon Master Bridge Cores chapter in volume 5 of the Quartus II Handbook
- AN 42: Metastability in Altera Devices
- Area and Timing Optimization chapter in volume 2 of the Quartus II Handbook

Document Revision History

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

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>Added a description to specify the shift direction.</td>
<td>—</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>Added description of a new parameter, Number of synchronizer stages: Depth.</td>
<td>—</td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td>—</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>Initial release.</td>
<td>—</td>
</tr>
</tbody>
</table>
Core Overview

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

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

This chapter contains the following sections:

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

Performance and Resource Utilization

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

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

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

<table>
<thead>
<tr>
<th>PCI Device Mode</th>
<th>PCI Target</th>
<th>PCI Master</th>
<th>ALUTs (2)</th>
<th>Logic Register</th>
<th>M9K Memory Blocks</th>
<th>I/O Pins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min (1)</td>
<td>Enabled</td>
<td>N/A</td>
<td>715</td>
<td>517</td>
<td>2</td>
<td>48</td>
</tr>
<tr>
<td>Max (1)</td>
<td>Enabled</td>
<td>Enabled</td>
<td>1,347</td>
<td>876</td>
<td>5</td>
<td>50</td>
</tr>
</tbody>
</table>

Notes to Table 11–1:
(1) Min = One BAR with minimum settings for each parameter.
Max = Three BARs with maximum settings for each parameter.
(2) The logic element (LE) count for the Stratix III family is based on the number of adaptive look-up tables (ALUTs) used for the design as reported by the Quartus II software.

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

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

<table>
<thead>
<tr>
<th>PCI Device Mode</th>
<th>PCI Target</th>
<th>PCI Master</th>
<th>Logic Elements</th>
<th>Logic Register</th>
<th>M4K Memory Blocks</th>
<th>I/O Pins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min (1)</td>
<td>Enabled</td>
<td>N/A</td>
<td>1,057</td>
<td>511</td>
<td>2</td>
<td>48</td>
</tr>
<tr>
<td>Max (1)</td>
<td>Enabled</td>
<td>Enabled</td>
<td>2,027</td>
<td>878</td>
<td>5</td>
<td>50</td>
</tr>
</tbody>
</table>

Note to Table 11–2:
(1) Min = One BAR with minimum settings for each parameter.
Max = Three BARs with maximum settings for each parameter.

Functional Description

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

PCI-Avalon Bridge Blocks

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

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

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

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

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

This section discusses the five Avalon-MM ports:

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

Prefetchable Avalon-MM Master

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

This port is optimized for high bandwidth transfers as a PCI target and it does not support single cycle transactions.
Non-Prefetchable Avalon-MM Master

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

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

I/O Avalon-MM Master

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

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

PCI Bus Access Slave

This Avalon-MM slave port propagates the following transactions from the interconnect to the PCI bus:
- Single cycle memory read and write requests
- Burst memory read and write requests
- I/O read and write requests
- Configuration read and write requests

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

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

Control Register Access (CRA) Avalon-MM Slave

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

Table 11–3 on page 11–5 shows the instructions on how to use these values. The address translation table is writable via the Control Register Access Avalon Slave port. If the Number of Address Pages field is set to the maximum of 512, 0x1F8 contains A2P_ADDR_MAP_LO511 and 0x1FFC contains A2P_ADDR_MAP_HI511.
Each entry in the PCI address translation table is always 8 bytes wide. The lower order address bits that are treated as a pass through between Avalon-MM and PCI, and the number of pass-through bits, are defined by the size of page in the address translation table and are always forced to 0 in the hardware table. For example, if the page size is 4 KBytes, the number of pass-through bits is \( \log_2 \text{(page size)} = \log_2 (4 \text{ KBytes}) = 12 \).

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

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

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

**Master and Target Performance**

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

**Master Performance**

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

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

**Target Performance**

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

**PCI-to-Avalon Address Translation**

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

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

Avalon-to-PCI Address Translation

Avalon-to-PCI address translation is done through a translation table. Low order Avalon-MM address bits are passed to PCI unchanged; higher order Avalon-MM address bits are used to index into the address translation table. The value found in the table entry is used as the higher order PCI address bits. Figure 11–3 depicts this process.
The address size selections in the translation table determine both the number of entries in the Avalon-to-PCI address translation table, and the number of bits that are passed through the transaction table unchanged.

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

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

Table 11–4. Address Space Bit Encodings

<table>
<thead>
<tr>
<th>Address Space Indicator (Bits 1:0)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Memory space, 32-bit PCI address. Address bits 63:32 of the translation table entries are ignored.</td>
</tr>
<tr>
<td>01</td>
<td>Reserved.</td>
</tr>
<tr>
<td>10</td>
<td>I/O space. The address from the translation table process is modified as described in Table 11–5.</td>
</tr>
<tr>
<td>11</td>
<td>Configuration space. The address from the translation table process is treated as a type 1 configuration address and is modified as described in Table 11–5.</td>
</tr>
</tbody>
</table>
If the space indication bits specify configuration or I/O space, subsequent modifications to the PCI address are performed. See Table 11–5.

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

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

Avalon-To-PCI Read and Write Operation

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Data Path Width</th>
<th>Avalon Burst Count</th>
<th>Type of Operation</th>
<th>Avalon Byte Enables</th>
<th>Resulting PCI Operation and Byte Enables</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1</td>
<td>Read or Write</td>
<td>Any value</td>
<td>Single data phase read or write, PCI byte enables identical to Avalon byte enables</td>
</tr>
<tr>
<td>32</td>
<td>&gt;1</td>
<td>Read</td>
<td>Any value</td>
<td>Attempt to burst on PCI. All data phases have all PCI bytes enabled.</td>
</tr>
<tr>
<td>32</td>
<td>&gt;1</td>
<td>Write</td>
<td>Any value</td>
<td>Attempt to burst on PCI. All data phases have PCI byte enables identical to the Avalon byte enables.</td>
</tr>
</tbody>
</table>
**Avalon-to-PCI Write Requests**

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

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

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

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

**Avalon-to-PCI Read Requests**

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

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

Read requests on PCI may initially be retried. Retries depend on the response time from the target. The master continues to retry until it gets the required data.
Table 11–8 shows PCI master read request termination conditions.

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

## Ordering of Requests

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

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

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

## PCI Interrupt

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

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

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

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Master/Target Mode</td>
<td>On or Off</td>
<td>Turning this option On enables Master/Target mode. This option enables allows Avalon-MM master devices to access PCI target devices via the PCI bus master interface, and PCI bus master devices to access Avalon-MM slave devices via the PCI bus target interface. Turning this option Off means you have selected Target Only mode, which allows PCI bus mastering devices to access Avalon-MM slave devices via the PCI bus target interface.</td>
</tr>
<tr>
<td>Enable Host Bridge Mode</td>
<td>On or Off</td>
<td>Turning this option On enables this mode. In addition to the same features provided by the PCI Master/Target mode, Host Bridge Mode provides host bridge functionality including hardwiring the master enable bit to 1 in the PCI command register and allowing self-configuration. This value can only be set if the Enable Master/Target Mode option is turned On.</td>
</tr>
<tr>
<td>Number of Address Pages</td>
<td>2, 4, 8, or 16</td>
<td>The number of translation/pages supported by the device for Avalon to PCI address translation.</td>
</tr>
<tr>
<td>Size of Address Pages</td>
<td>12–27</td>
<td>The supported address size (in bits) that can be assigned to each map number entries.</td>
</tr>
<tr>
<td>Prefetchable BAR</td>
<td>On or Off</td>
<td>Turning this option On invokes a Prefetchable Master (PM) Bar in the PCI system. This option allows PCI-Avalon Bridge Lite to accept and process PM transactions.</td>
</tr>
<tr>
<td>Prefetchable BAR Size</td>
<td>10–31</td>
<td>The allowed reserved address range supported by the PM BAR. The reserved memory space is 1 KByte (10 bits) to 4 GBytes (31 bits).</td>
</tr>
<tr>
<td>Prefetchable BAR Avalon Address Translation Offset</td>
<td>&lt;BAR translation value&gt;</td>
<td>The direct translation of the value that hits the BAR and modified to a fixed address in the Avalon space. Refer to “PCI-to-Avalon Address Translation” on page 11–6.</td>
</tr>
<tr>
<td>Non-Prefetchable BAR</td>
<td>On or Off</td>
<td>Turning this option On invokes a Non-Prefetchable Master (NPM) Bar in the PCI system. This option allows the PCI-Avalon Bridge Lite to accept and process NPM transactions.</td>
</tr>
<tr>
<td>Non-Prefetchable BAR Size</td>
<td>10–31</td>
<td>Specifies the allowed reserved address range supported by the NPM BAR. The reserved memory space is 1 KByte (10 bits) to 4 GBytes (31 bits).</td>
</tr>
<tr>
<td>Non-Prefetchable BAR Avalon Address Translation Offset</td>
<td>&lt;BAR translation value&gt;</td>
<td>The direct translation of the value that hits the BAR and modified to a fixed address in the Avalon space. Refer to “PCI-to-Avalon Address Translation” on page 11–6.</td>
</tr>
<tr>
<td>I/O BAR</td>
<td>On or Off</td>
<td>Turning this option On enables an I/O BAR in the system. This option allows PCI-Avalon Bridge Lite to accept and process I/O type transactions.</td>
</tr>
<tr>
<td>I/O BAR Size</td>
<td>2–8</td>
<td>The allowed reserved address range supported by the I/O BAR. The reserved memory space is 4 bytes (2 bits) to 256 bytes (8 bits).</td>
</tr>
<tr>
<td>I/O BAR Avalon Address Translation Offset</td>
<td>&lt;BAR translation value&gt;</td>
<td>The direct translation of the value that hits the BAR and modified to a fixed address in the Avalon address space. Refer to “PCI-to-Avalon Address Translation” on page 11–6.</td>
</tr>
</tbody>
</table>
Table 11–9. Parameters for PCI Lite Core (Part 2 of 2)

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

**PCI Timing Constraint Files**

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

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


Quartus II Handbook Version 9.1 Volume 5: Embedded Peripherals © November 2009 Altera Corporation
2. Update the pin list in the Tcl constraint file. Edit the `get_user_pin_name` procedure in the Tcl constraint file to match the default pin names. To edit the PCI constraint file, follow these steps:
   a. Locate the `get_user_pin_name` procedure. This procedure maps the default PCI pin names to user PCI pin names. The following lines are the first few lines of the procedure:

   ```tcl
   proc get_user_pin_name { internal_pin_name } {
   #---------------- Do NOT change ------------------------------- ---- Change -----
   array set map_user_pin_name_to_internal_pin_name {ad ad }
   
   b. Edit the pin names under the Change header in the file to match the PCI pin names used in your Quartus II project. In the following example, the name ad is changed to pci_ad:

   ```tcl
   array set map_user_pin_name_to_internal_pin_name { ad pci_ad }
   ```

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

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

   ```tcl
   source pci_constraints.tcl
   ```

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

   ```tcl
   add_pci_constraints
   ```

   See “Additional Tcl Option” on page 11–13 for the option supported by the `add_pci_constraints` command.

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

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

Additional Tcl Option

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

```tcl
add_pci_constraints [-no_compile]
```

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

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

Simulation Considerations

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

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

Features

The PCI testbench includes the following features:

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

Master Transactor (mstr_tranx)

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

- TASKS (Verilog HDL)
- INITIALIZATION
- USER COMMANDS

TASKS Sections

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

- Address phase
- Turn-around phase (read transactions)
- Data phases
- Turn-around phase
The master transactor terminates the PCI transactions in the following cases:

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

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

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

**INITIALIZATION Section**

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

**USER COMMANDS Section**

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

**Simulation Flow**

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

**Figure 11–4. Typical Verification Environment Using the PCI Testbench**
The simulation flow using Altera PCI testbench comprises the following steps.

1. Use SOPC Builder to create your system. SOPC creates the `<variation name_system>_sim` folder in your project directory.

2. Source `pci_constraints.tcl`.

3. Copy

   `<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite/rtgt_tranx_mem_init.dat` to `<project_directory>/<variation name_system>_sim` folder.

4. Edit the top level HDL verilog files in the testbench. Insert the following lines just before `module test_bench`.

   ```
   `include "<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite/pci_tb.v"
   `include "<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite/clk_gen.v"
   `include "<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite/arbiter.v"
   `include "<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite/pull_up.v"
   `include "<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite/monitor.v"
   `include "<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite/trgt_tranx.v"
   `include "mstr_tranx.v"
   ```

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

5. Set the initialization parameters, which are defined in the master transactor model source code. These parameters control the address space reserved by the target transactor model and other PCI agents on the PCI bus.

6. The master transactor defines the tasks (Verilog HDL) needed to initiate PCI transactions in your testbench. Add the commands that correspond to the transactions you want to implement in your tests to the master transactor model source code. At a minimum, you must add configuration commands to set the BAR for the target transactor model and write the configuration space of the PCI Lite core. Additionally, you can add commands to initiate memory or I/O transactions to the PCI Lite core.
7. Compile the files in your simulator, including the testbench modules and the files created by SOPC Builder.

8. Simulate the testbench for the desired time period.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 11–10 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. Edited the command errors in the Simulation Flow section.</td>
<td>—</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>Initial release.</td>
<td>—</td>
</tr>
</tbody>
</table>
Core Overview

The Cyclone® III Remote Update Controller core provides a method to control the Cyclone III remote update block from SOPC Builder systems. The core allows you to access all features of the ALTREMOTE_UPDATE megafunction through a simple Avalon® Memory-Mapped (Avalon-MM) slave interface. The slave interface allows Avalon-MM master peripherals, such as a Nios® II processor, to communicate with the core simply by reading and writing the registers.

The Cyclone III Remote Update Controller core is a thin Avalon interface layer on top of the ALTREMOTE_UPDATE megafunction. Every function of the core maps directly to a function of the megafunction. Altera recommends that you familiarize yourself with the ALTREMOTE_UPDATE megafunction before using the core.

For more information about the ALTREMOTE_UPDATE megafunction, refer to the altremote_update Megafunction User Guide. For more information about remote system upgrade in Cyclone III devices, refer to the Remote System Upgrade With Cyclone III Devices chapter in volume 1 of the Cyclone III Device Handbook.

The Cyclone III Remote Update Controller core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- “Functional Description”
- “Device Support” on page 12–2
- “Instantiating the Core in SOPC Builder” on page 12–2

Functional Description

Figure 12–1 shows a block diagram of the Cyclone III Remote Update Controller core.

![Cyclone III Remote Update Controller Core Block Diagram](image)
Avalon-MM Slave Interface and Registers

The address bus on the core’s Avalon-MM interface is 6 bits wide. The lower three bits of the address bus map directly to the param signal of the ALTREMOTE_UPDATE megafunction whereas the upper three bits map to the read_source signal.

Reading or writing to address offsets 0x00 – 0x1F of the Cyclone III Remote Update Controller core is equivalent to performing read or write operations to the ALTREMOTE_UPDATE megafunction using the param and read_source signals.

Table 12–1 shows the mapping of the 5 lowest order Remote Update Controller address bits to the ALTREMOTE_UPDATE megafunction signals.

Table 12–1. Avalon-MM Address Bits to Megafunction Signals Mapping

<table>
<thead>
<tr>
<th>Address Bit</th>
<th>Megafunction Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>address[0]</td>
<td>param[0]</td>
</tr>
<tr>
<td>address[1]</td>
<td>param[1]</td>
</tr>
<tr>
<td>address[3]</td>
<td>read_source[0]</td>
</tr>
</tbody>
</table>

The highest order address bit [5] is used to access a single control/status register. Reading or writing any address offset from 0x20 – 0x3F accesses the control/status register.

Table 12–2 shows the bit map of the control/status register.

Table 12–2. Bit Map of Control/Status Register

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RECONFIG</td>
<td>RW</td>
<td>Set this bit to 1 to reset the FPGA and trigger reconfiguration.</td>
</tr>
<tr>
<td>1</td>
<td>RESET_TIMER</td>
<td>RW</td>
<td>Set this bit to 1 to reset the watchdog timer. Then, set this bit to 0 to allow the watchdog timer to continue.</td>
</tr>
<tr>
<td>2..31</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Device Support

The Cyclone III Remote Update Controller core can only target Cyclone III device family. Both CFI flash and EPCS configuration devices are supported as non-volatile storage for configuration images.

Instantiating the Core in SOPC Builder

The Cyclone III Remote Update Controller core has no user-configurable parameters.
Software Programming Model

Software programs can operate the Cyclone III Remote Update Controller core by reading from and writing to the core’s registers.

You can only reconfigure the FPGA to an application image from the factory image. Any attempt to reconfigure from an already reconfigured application image causes the FPGA to return to the factory image.

This section describes the most common types of operations using the Cyclone III Remote Update Controller core.

Setting the Configuration Offset

Before you reconfigure the FPGA, you must first specify the offset within the memory device from which you want to execute a reconfiguration. The offset is the relative address within the memory device where the configuration image is located. Write the offset value to address 0x04 of the core to set the configuration offset.

For example, if your system contains a CFI flash memory mapped at address 0x04000000, and the configuration image is located at address 0x100000 in the flash memory, the offset to set in the Cyclone III Remote Update Controller core is 0x100000.

Shifting the Configuration Offset Value

The ALTREMOTE_UPDATE megafunction requires that you provide only the 22 highest-order bits of a 24-bit address offset. To translate the address, right shift the offset by two bits. This results in a properly oriented 22-bit address offset.

If you are using a CFI flash device, you must also take into account the data width of the flash. If the data width of your flash device is 16 bits, you must provide a 16-bit address offset to the Cyclone III Remote Update Controller core. This requires an additional 1-bit right shift of the byte address offset. No translation is necessary if the data width of your flash is 8 bits.

If you are using an EPCS serial configuration device, consider the data width of the device to be 8 bits. Even though the EPCS device is a serial device, it uses byte addressing internally.

For example, an FPGA is set up to configure itself using active parallel mode from a 16-bit CFI flash memory mapped at address 0x04000000 in an SOPC Builder system, and the configuration image is located at byte offset 0x100000 within the flash memory. To derive the correct configuration offset, you must first right shift the byte offset 0x100000 by one bit to obtain the 16-bit address. Then, right shift by another two bits to obtain the highest 22 bits of the 24-bit offset. The result is a configuration offset of 0x20000 (0x100000 >> 3 = 0x20000), to be written to address 0x04 of the core.

Setting up the Watchdog Timer

You can set up the watchdog timer by writing the upper 12 bits of the 29-bit timeout value to address 0x02 of the core. To reset the watchdog timer, set the RESET_TIMER bit of the control/status register to 1 and immediately set the bit to 0.
Ensure that you don’t accidentally set bit 0 of the control/status register to 1. Otherwise, you will trigger a reconfiguration of the FPGA.

For more information on watchdog timer, refer to the ALTREMOTE_UPDATE Megafuction User Guide.

If you do not use the watchdog timer feature of the ALTREMOTE_UPDATE megafunction, it must be disabled before a reconfiguration is performed. To disable the watchdog timer, write 0x00 to address 0x03 of the core.

**Triggering a Reconfiguration**

You can trigger a reconfiguration once you have set the reconfiguration offset in the Cyclone III Remote Update Controller core, and you have either setup or disabled the watchdog timer. To trigger a reconfiguration, set the RECONFIG bit in the control/status register to 1. Consequently, the FPGA performs a reset and reconfigures itself from the configuration image specified.
Code Example

Example 12–1 shows a C function that can be used to operate the Cyclone III Remote Update Controller core from a processor such as Nios II.

**Example 12–1. FPGA Reconfiguration Function**

```c
int CycloneIII_Reconfig( int remote_update_base,
                          int flash_base,
                          int reconfig_offset,
                          int watchdog_timeout,
                          int width_of_flash )
{
    int offset_shift;

    // Obtain upper 12 bits of 29-bit watchdog timeout value
    watchdog_timeout = watchdog_timeout >> 17;

    // Only enable the watchdog timer if its timeout value is greater than 0.
    if( watchdog_timeout > 0 )
    {
        // Set the watchdog timeout value
        IOWR( remote_update_base, 0x2, watchdog_timeout );
    }
    else
    {
        // Disable the watchdog timer
        IOWR( remote_update_base, 0x3, 0 );
    }

    // Calculate how much to shift the reconfig offset location:
    // width_of_flash == 8->offset_shift = 2.
    // width_of_flash == 16->offset_shift = 3
    offset_shift = ( ( width_of_flash / 8 ) + 1 );

    // Write the offset of the desired reconfiguration image in flash
    IOWR( remote_update_base, 0x4, reconfig_offset >> offset_shift );

    // Perform the reconfiguration by setting bit 0 in the
    // control/status register
    IOWR( remote_update_base, 0x20, 0x1 );

    return( 0 );
}
```
Related Documentation

This chapter references the following documents:

- altremote_update Megafuinction User Guide

Document Revision History

Table 12–3 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>No change from previous release.</td>
<td>—</td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Initial release.</td>
<td>—</td>
</tr>
</tbody>
</table>