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

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

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

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

Figure 5–1. JTAG UART Core Block Diagram

Avalon Slave Interface and Registers

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

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

Read and Write FIFOs

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

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

**Host-Target Connection**

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

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

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

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

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

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

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

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

Instantiating the Core in SOPC Builder

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

Configuration Page

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

Write FIFO Settings

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

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

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

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

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

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

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

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

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

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

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

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

Hardware Simulation Considerations

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

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

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

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

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

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

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

```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.
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 5–1. 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*. 
Table 5–2. JTAG UART Core Register Map

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

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

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

Table 5–3. data Register Bits

<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>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<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 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.
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.**

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

**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.</td>
</tr>
</tbody>
</table>

- 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 Odd, 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 Even, the parity bit is 0 if the character has an odd number of 1 bits.

Synchronizer Stages

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

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

Flow Control

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

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

When the Include CTS/RTS pins and control register bits setting is off, the core does not include the aforementioned hardware and continuous writes to the UART may lose 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')
            {
                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 6–2. UART ioctl() Operations**

<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 <code>TIOCNXCL 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>

Note to Table 8–3:

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

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

**Limitations**

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

**Software Files**

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

- `altera_aavalon_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_aavalon_uart.h, altera_aavalon_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 iibrk 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 exists in hardware only if it was enabled at system generation time. Optional registers and bits are noted in the following sections.

**rxdata Register**

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

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

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

status Register

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

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

<table>
<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>FE</td>
<td>RC</td>
<td>Framing error. A framing error occurs when the receiver fails to detect a correct stop bit. The FE bit is set to 1 when the core receives a character with an incorrect stop bit. The FE bit stays set to 1 until it is explicitly cleared by a write to the status register. When the FE bit is set, reading from the rxdata register produces an undefined value.</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>ITRMT</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</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
Document Revision History

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>March 2009 v9.0.0</td>
<td>Added description of a new parameter, Synchronizer stages.</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.

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.

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 1 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 1 to 16. A master peripheral reads received data from the \( rxdata \) register after the shift register has captured a full \( n \)-bit value of data.

The shift register and the \( rxdata \) register provide double buffering 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.

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.

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sclk</td>
<td>output</td>
<td>Synchronization clock to all slaves</td>
</tr>
<tr>
<td>ss_nM</td>
<td>output</td>
<td>Slave select signal to slave M, where M is a number between 0 and 15.</td>
</tr>
</tbody>
</table>
Multi-Slave Environments

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

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

Avalon-MM Interface

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

Instantiating the SPI Core in SOPC Builder

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

Master/Slave Settings

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

**Number of Select (SS_n) Signals**

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

**SPI Clock (sclk) Rate**

This setting determines the rate of the sclk signal that synchronizes data between master and slaves. The target clock rate can be specified in units of Hz, kHz or MHz. The SPI master core uses the Avalon-MM system clock and a clock divisor to generate sclk.
The actual frequency of \( sclk \) may not exactly match the desired target clock rate. The achievable clock values are:

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

The actual frequency achieved will not be greater than the specified target value. For example, if the system clock frequency is 50 MHz and the target value is 25 MHz, 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, \( \mu s \) or ms. An example is shown in Figure 7–4.

**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 \texttt{ss\_n}, \texttt{sclk}, \texttt{mosi} and \texttt{miso} signals. In this discussion the \texttt{mosi} and \texttt{miso} signals are referred to generically as \texttt{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 \texttt{sclk} is low. When clock polarity is set to 1, the idle state for \texttt{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 \texttt{sclk}, and data changes on trailing edge. When clock phase is 1, data is latched on the trailing edge of \texttt{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

- \texttt{SS\_n}
- \texttt{SCLK}
- \texttt{DATA\_OUT} MSB LSB

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

- \texttt{SS\_n}
- \texttt{SCLK}
- \texttt{DATA\_OUT} MSB LSB

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

- \texttt{SS\_n}
- \texttt{SCLK}
- \texttt{DATA\_OUT} MSB LSB

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

- \texttt{SS\_n}
- \texttt{SCLK}
- \texttt{DATA\_OUT} MSB LSB
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.
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 reads 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.

### Table 7-3. Register Map for SPI Master Device

<table>
<thead>
<tr>
<th>Internal Address</th>
<th>Register Name</th>
<th>32..11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rxdata (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>txdata (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>status (2)</td>
<td></td>
<td></td>
<td>E</td>
<td>RRDY</td>
<td>TRDY</td>
<td>TMT</td>
<td>TOE</td>
<td>ROE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>control</td>
<td>SSO</td>
<td></td>
<td>IE</td>
<td>IRRDY</td>
<td>ITRDY</td>
<td>ITOE</td>
<td>IROE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>slaveselect (3)</td>
<td>Slave Select Mask</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes to Table 7-3:**

1. Bits 15 to \( n \) are undefined when \( n \) is less than 16.
2. A write operation to the status register clears the ROE, TOE, and E bits.
3. Present only in master mode.
As an example, assume that the SPI core is idle (that is, the txdata register and transmit shift register are empty), when a CPU writes a data value into the txdata holding register. The TRDY bit is set to 0 momentarily, but after the data in txdata is transferred into the transmitter shift register, TRDY returns to 1. The CPU writes a second data value into the txdata register, and again the TRDY bit is set to 0. This time the shift register is still busy transferring the original data value, so the TRDY bit remains at 0 until the shift operation completes. When the operation completes, the second data value is transferred into the transmitter shift register and the TRDY bit is again set to 1.

**status Register**

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

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>ROE</td>
<td><strong>Receive-overrun error</strong>&lt;br&gt;The ROE bit is set to 1 if new data is received while the rxdata register is full (that is, while the RRDY bit is 1). In this case, the new data overwrites the old. Writing to the status register clears the ROE bit to 0.</td>
</tr>
<tr>
<td>4</td>
<td>TOE</td>
<td><strong>Transmitter-overrun error</strong>&lt;br&gt;The TOE bit is set to 1 if new data is written to the txdata register while it is still full (that is, while the TRDY bit is 0). In this case, the new data is ignored. Writing to the status register clears the TOE bit to 0.</td>
</tr>
<tr>
<td>5</td>
<td>TMT</td>
<td><strong>Transmitter shift-register empty</strong>&lt;br&gt;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.&lt;br&gt;In slave mode, the TMT bit is set to 0 when the slave is selected (SS_n is low) or when the SPI Slave register interface is not ready to receive data.</td>
</tr>
<tr>
<td>6</td>
<td>TRDY</td>
<td><strong>Transmitter ready</strong>&lt;br&gt;The TRDY bit is set to 1 when the txdata register is empty.</td>
</tr>
<tr>
<td>7</td>
<td>RRDY</td>
<td><strong>Receiver ready</strong>&lt;br&gt;The RRDY bit is set to 1 when the rxdata register is full.</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td><strong>Error</strong>&lt;br&gt;The E bit is the logical OR of the TOE and ROE bits. This is a convenience for the programmer to detect error conditions. Writing to the status register clears the E bit to 0.</td>
</tr>
</tbody>
</table>
control Register

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

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

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

### Table 7–5. control Register Bits

<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>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 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 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.
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.

**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>
<thead>
<tr>
<th>Table 8–1. Escape Sequence Supported by the LCD Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
</tr>
<tr>
<td>BS  (\b)</td>
</tr>
<tr>
<td>CR  (\r)</td>
</tr>
<tr>
<td>LF  (\n)</td>
</tr>
<tr>
<td>ESC\x1B</td>
</tr>
<tr>
<td>ESC [ y ; x H</td>
</tr>
<tr>
<td>ESC [ K</td>
</tr>
<tr>
<td>ESC [ 2 J</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>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*. 
9. PIO Core

Core Overview

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

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

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

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

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

Functional Description

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

![Block diagram of PIO core with input and output ports and IRQ support](image1)

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

![Block diagram of PIO core in bidirectional mode](image2)

**Avalon-MM Interface**

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

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

Basic Settings

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

**Width**

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

**Direction**

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

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

**Output Port Reset Value**

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

**Output Register**

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

**Input Options**

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

**Edge Capture Register**

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

- Rising Edge
- Falling Edge
- Either Edge

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

**Interrupt**

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

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

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

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

**Simulation**

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

**Device Support**

The PIO core supports all Altera® device families.

**Software Programming Model**

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

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

**Software Files**

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

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>R/W (n-1)</th>
<th>...</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>data</td>
<td>read access</td>
<td>R</td>
<td>Data value currently on PIO inputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>write access</td>
<td>W</td>
<td>New value to drive on PIO outputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>direction ((1))</td>
<td>R/W</td>
<td>Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>interruptmask ((1))</td>
<td>R/W</td>
<td>IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>edgecapture ((1), (2))</td>
<td>R/W</td>
<td>Edge detection for each input port.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>outset</td>
<td>W</td>
<td>Specifies which bit of the output port to set.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>outclear</td>
<td>W</td>
<td>Specifies which output bit to clear.</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>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<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.