7. JTAG UART Core

NII51009-7.1.0

Core Overview

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

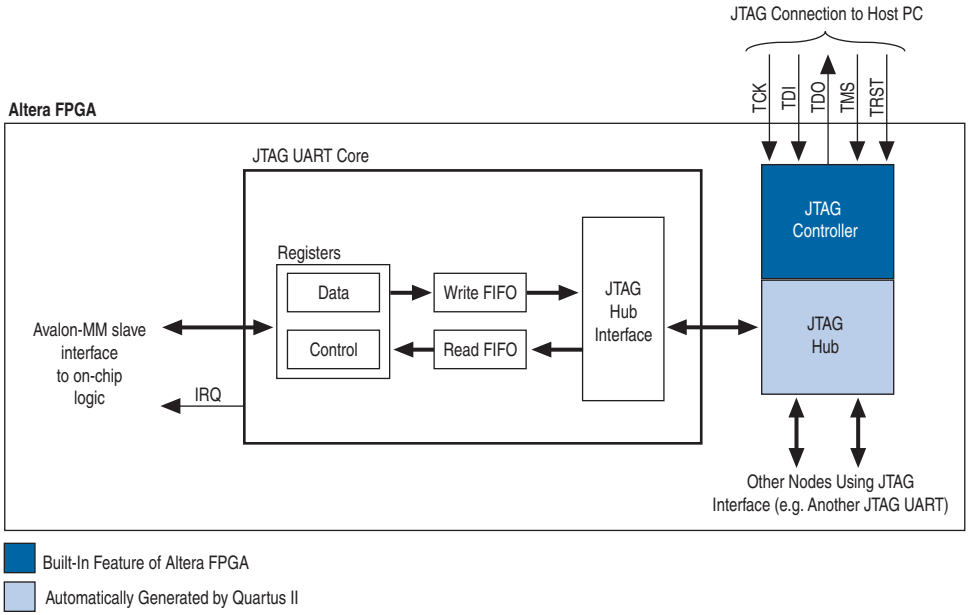
The JTAG UART core uses the JTAG circuitry built in to Altera FPGAs, and provides host access via the JTAG pins on the FPGA. The host PC can connect to the FPGA via any Altera JTAG download cable, such as the USB-Blaster™ cable. Software support for the JTAG UART core is provided by Altera. For the Nios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

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

Functional Description

Figure 7–1 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 7–1. JTAG UART Core Block Diagram



Avalon Slave Interface & Registers

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

The JTAG UART core provides an active-high interrupt output that can request an interrupt when read data is available, or when the write FIFO is ready for data. For further details see "Interrupt Behavior" on page 7–14.

Read & Write FIFOs

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

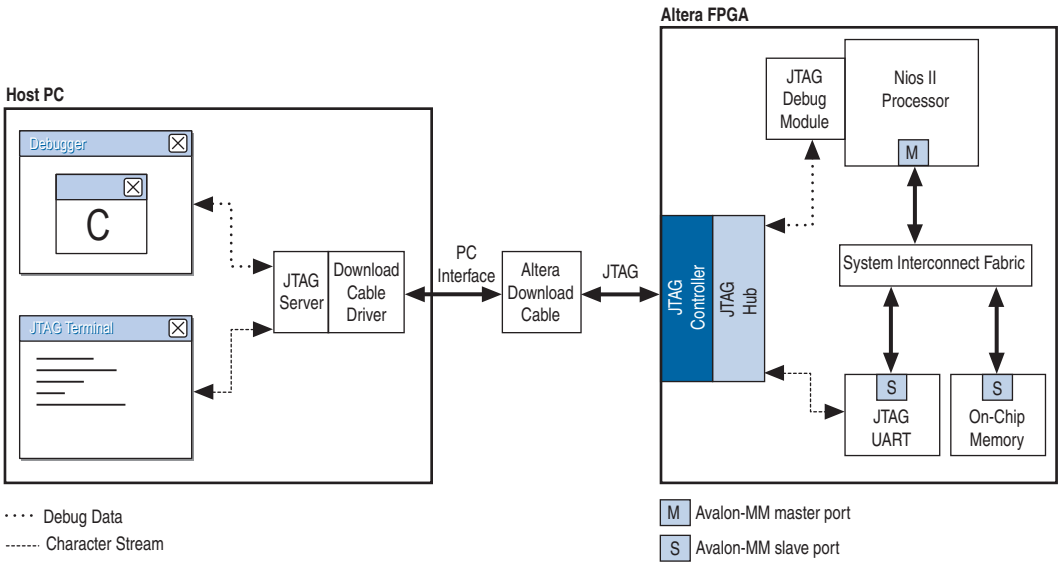
JTAG Interface

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

Host-Target Connection

Figure 7–2 shows the connection between a host PC and an SOPC Builder-generated system containing a JTAG UART core.

Figure 7–2. 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 7–2 contains one JTAG UART core and a Nios II processor. Both agents communicate to the host PC over a single Altera download cable. Thanks to the JTAG server software, each host application has an independent connection to the target. Altera provides the JTAG server drivers and host software required to communicate with the JTAG UART core.

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

Device Support & Tools

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

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

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

Instantiating the Core in SOPC Builder

Designers use the JTAG UART core's SOPC Builder configuration wizard to specify the core features. The following sections describe the available options in the configuration wizard.

Configuration Tab

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

Write FIFO Settings

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

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

- **IRQ Threshold**—The write IRQ threshold governs how the core asserts its IRQ in response to the FIFO emptying. As the JTAG circuitry empties data from the write FIFO, the core asserts its IRQ when the number of characters remaining in the FIFO reaches this threshold value. For maximum bandwidth efficiency, a processor should service the interrupt by writing more data and preventing the write FIFO from emptying completely. A value of 8 is typically optimal. See "Interrupt Behavior" on page 7–14 for further details.

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

Read FIFO Settings

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

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

- **IRQ Threshold—**The IRQ threshold governs how the core asserts its IRQ in response to the FIFO filling up. As the JTAG circuitry fills up the read FIFO, the core asserts its IRQ when the amount of space remaining in the FIFO reaches this threshold value. For maximum bandwidth efficiency, a processor should service the interrupt by reading data and preventing the read FIFO from filling up completely. A value of 8 is typically optimal. See "Interrupt Behavior" on page 7–14 for further details.

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

Simulation Settings

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

Simulated Input Character Stream

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

Prepare Interactive Windows

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

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

the console are fed into the read FIFO, and can be read via the Avalon interface. When this option is enabled, the simulated character input stream option is ignored.

Hardware Simulation Considerations

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

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

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

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

Software Programming Model

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

HAL System Library Support

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

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

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

"Printing Characters to a JTAG UART Core as stdout" demonstrates the simplest possible usage, printing a message to stdout using `printf()`. In this example, the SOPC Builder system contains a JTAG UART core, and the HAL system library has been configured to use this JTAG UART device for stdout.

Printing Characters to a JTAG UART Core as stdout

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

"Transmitting Characters to a JTAG UART Core" on page 7–9 demonstrates reading characters from and sending messages to a JTAG UART core using the C standard library. In this example, the SOPC Builder system contains a JTAG UART core named `jtag_uart` that is not necessarily configured as the stdout device. In this case, the program treats the device like any other node in the HAL file system.

Transmitting Characters to a JTAG UART Core

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

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

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

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

  return 0;
}
```

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

The *Nios II Software Developer's Handbook* provides complete details of the HAL system library. The Nios II 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 provides two variants: A fast version and a small version. The fast behavior will be used by default. Both the fast and small drivers fully support the C standard library functions and the HAL API.

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

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

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

ioctl() Operations

The fast version of the JTAG UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. Specifically, you can use the `ioctl()` operations to control the timeout period, and to detect whether or not a host is connected. The fast driver defines the `ioctl()` operations shown in Table 7–1.

Table 7–1. JTAG UART ioctl() Operations for the Fast Driver Only

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

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

Software Files

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

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

Accessing the JTAG UART Core via a Host PC

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

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

Register Map

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

CAUTION

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

Table 7–2 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 7–2. JTAG UART Core Register Map

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

Note to Table 7–2:
(1) Reserved. Read values are undefined. Write zero.

Data Register

Embedded software accesses the read and write FIFOs via the `data` register. Table 7–3 describes the function of each bit.

Table 7–3. data Register Bits

Bit Number	Bit/Field Name	Read/Write/Clear	Description
0 .. 7	DATA	R/W	The value to transfer to/from the JTAG core. When writing, the DATA field is a character to be written to the write FIFO. When reading, the DATA field is a character read from the read FIFO.
15	RVALID	R	Indicates whether the DATA field is valid. If RVALID=1, then the DATA field is valid, else DATA is undefined.
16 .. 32	RAVAIL	R	The number of characters remaining in the read FIFO (after this read).

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

Control Register

Embedded software controls the JTAG UART core's interrupt generation and reads status information via the `control` register. Table 7–4 describes the function of each bit.

Table 7–4. control Register Bits

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

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

The RE and WE bits enable interrupts for the read and write FIFOs, respectively. The WI and RI bits indicate the status of the interrupt sources, qualified by the values of the interrupt enable bits (WE and RE). Embedded software can examine RI and WI to determine what condition generated the IRQ. See "Interrupt Behavior" on page 7–14 for further details.

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

Interrupt Behavior

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

Interrupt behavior is of concern to device driver programmers concerned with the bandwidth performance to the host PC. Example designs and the JTAG terminal program provided with Nios II 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 will be provided (or consumed) by the host PC every 1µs. With a threshold of 8, the interrupt response time must be less than 8µs. If the interrupt response time is too long, then performance will suffer. If it is too short, then interrupts will occur too frequently.

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

Document Revision History

Table 7–5 shows the revision history for this chapter.

Table 7–5. Document Revision History

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