8. UART Core

NII51010-7.1.0

Core Overview

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

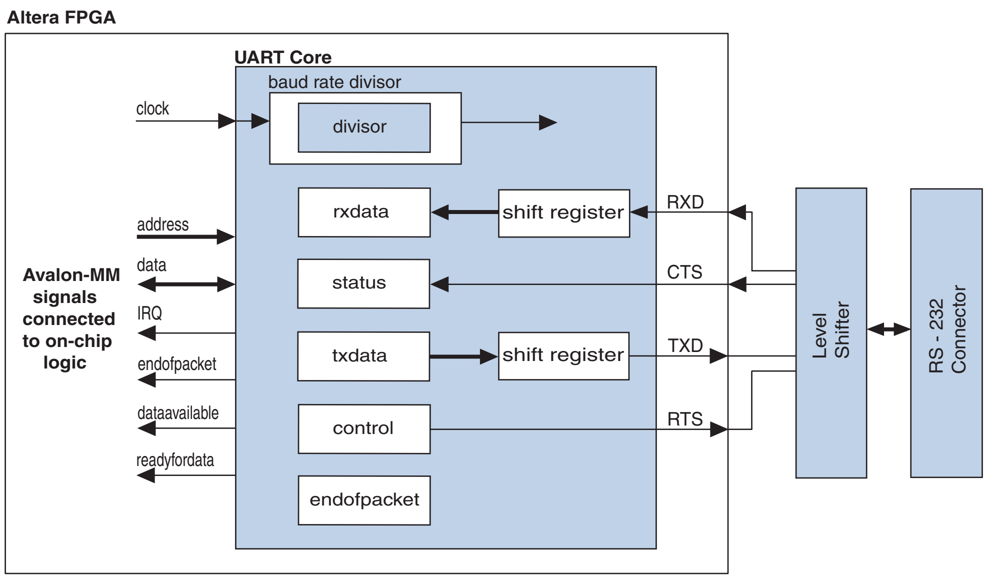
The core provides a simple register-mapped Avalon Memory-Mapped (Avalon-MM) slave interface that allows Avalon-MM master peripherals (such as a Nios® II processor) to communicate with the core simply by reading and writing control and data registers.

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

Functional Description

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

Figure 8–1. 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 see "Interrupt Behavior" on page 8–20.

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

See the *Timer Core* chapter for details. See the *Avalon Memory-Mapped Interface Specification* for details of the Avalon-MM interface.

RS-232 Interface

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

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

Transmitter Logic

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

These two registers provide double buffering. A master peripheral can write a new value into the `txdata` register while the previously written character is being shifted out. The master peripheral can monitor the transmitter's status by reading the `status` register's transmitter ready (trdy), transmitter shift register empty (tmt), and transmitter overrun error (toe) bits.

The transmitter logic automatically inserts the correct number of start, stop, and parity bits in the serial `TXD` data stream as required by the RS-232 specification.

Receiver Logic

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

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

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

Baud Rate Generation

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

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

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

Device and Tools Support

The UART core can target all Altera FPGAs.

Instantiating the Core in SOPC Builder

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

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

Configuration Settings

This section describes the configuration settings.

Baud Rate Options

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

- **Fixed rate**—The baud rate is fixed at system generation time and cannot be changed via the Avalon-MM slave port.
- **Variable rate**—The baud rate can vary, based on a clock divisor value held in the `divisor` register. A master peripheral changes the baud rate by writing new values to the `divisor` register.

☞ The baud rate is calculated based on the clock frequency provided by the Avalon-MM interface. Changing the system clock frequency in hardware without re-generating the UART core hardware will result in incorrect signaling.

Baud Rate (bps) Setting
The **Baud Rate** setting determines the default baud rate after reset. The **Baud Rate** option offers standard preset values (e.g., 9600, 57600, 115200 bps), or you can manually enter any baud rate.

The baud rate value is used to calculate an appropriate clock divisor value to implement the desired baud rate. Baud rate and divisor values are related as follows:

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

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

Baud Rate Can Be Changed By Software Setting
When this setting is on, the hardware includes a 16-bit `divisor` register at address offset 4. The `divisor` register is writable, so the baud rate can be changed by writing a new value to this register.

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

Data Bits, Stop Bits, Parity

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

Data Bits Setting
See Table 8–1.

Table 8–1. Data Bits Setting

Setting	Allowed Values	Description
Data Bits	7, 8, 9	This setting determines the widths of the `txdata`, `rxdata`, and `endofpacket` registers.
Stop Bits	1, 2	This setting determines whether the core transmits 1 or 2 stop bits with every character. The core always terminates a receive transaction at the first stop bit, and ignores all subsequent stop bits, regardless of the Stop Bits setting.
Parity	None, Even, Odd	This setting determines whether the UART transmits characters with parity checking, and whether it expects received characters to have parity checking. See below for further details.

Parity Setting
When **Parity** is set to **None**, the transmit logic sends data without including a parity bit, and the receive logic presumes the incoming data does not include a parity bit. When parity is None, the status register's pe (parity error) bit is not implemented; it always reads 0.

When **Parity** is set to **Odd** or **Even**, the transmit logic computes and inserts the required parity bit into the outgoing `TXD` bitstream, and the receive logic checks the parity bit in the incoming `RXD` bitstream. If the receiver finds data with incorrect parity, the status register's pe is set to 1. When parity is Even, the parity bit is 0 if the character has an even number of 1 bits; otherwise the parity bit is 1. Similarly, when parity is Odd, the parity bit is 0 if the character has an odd number of 1 bits.

Flow Control

The following flow control option is available.

Include CTS/RTS pins & control register bits
When this setting is on, the UART hardware includes:

- CTS_N (logic negative `CTS`) input port
- RTS_N (logic negative `RTS`) output port
- CTS bit in the `status` register

- DCTS bit in the `status` register
- RTS bit in the `control` register
- IDCTS bit in the `control` register

Based on these hardware facilities, an Avalon-MM master peripheral can detect `CTS` and transmit RTS flow control signals. The `CTS` input and RTS output ports are tied directly to bits in the `status` and `control` registers, and have no direct effect on any other part of the core.

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

Avalon-MM Transfers With Flow Control (DMA)

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

Include end-of-packet register
When this setting is on, the UART core includes:

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

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

When the end-of-packet register is disabled, the UART core does not include the resources listed above. Writing to the `endofpacket` register has no effect, and reading produces an undefined value.

Simulation Settings

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

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

Simulated RXD-Input Character Stream

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

Prepare Interactive Windows

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

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

Create ModelSim Alias to open interactive stimulus window
A ModelSim macro is created to open a window that accepts stimulus for the `RXD` port. The window sends any characters typed in the window to the `RXD` port.

Simulated Transmitter Baud Rate

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

- **accelerated (use divisor = 2)**—`TXD` emits one bit per 2 clock cycles in simulation.
- **actual (use true baud divisor)**—`TXD` transmits at the actual baud rate, as determined by the `divisor` register.

Hardware Simulation Considerations

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

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

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

For details about simulating the UART core in Nios II processor systems see *AN 351: Simulating Nios II Processor Designs*. For details about simulating the UART core in Nios embedded processor systems see *AN 189: Simulating Nios Embedded Processor Designs.*

Software Programming Model

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

HAL System Library Support

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

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

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

The driver supports the CTS/RTS control signals when they are enabled in SOPC Builder. See "Driver Options: Fast Versus Small Implementations" on page 8–11.

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

Example: Printing Characters to a UART Core as stdout

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

The following code demonstrates reading characters from and sending messages to a UART device using the C standard library. In this example, the SOPC Builder system contains a UART core named `uart1` that is not necessarily configured as the stdout device. In this case, the program treats the device like any other node in the HAL file system.

Example: Sending and Receiving Characters

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

  fp = fopen ("/dev/uart1", "r+"); //Open file for reading and writing
  if (fp)
  {
    while (prompt != 'v')
    {  // Loop until we receive a 'v'.
      prompt = getc(fp);  // Get a character from the UART.
      if (prompt == 't')
      {  // Print a message if character is 't'.
        fwrite (msg, strlen (msg), 1, fp);
      }
    }

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

  return 0;
}
```

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

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 behavior will be used by default. Both the fast and small drivers fully support the C standard library functions and the HAL API.

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

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

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

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

If the CTS/RTS flow control signals are enabled in hardware, the fast driver automatically uses them. The small driver always ignores them.

ioctl() Operations

The UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. Table 8–2 defines operation requests that the UART driver supports.

Table 8–2. UART ioctl() Operations

Request	Meaning
`TIOCEXCL`	Locks the device for exclusive access. Further calls to `open()` for this device will fail until either this file descriptor is closed, or the lock is released using the TIOCNXCL `ioctl` request. For this request to succeed there can be no other existing file descriptors for this device. The `ioctl` "arg" parameter is ignored.
`TIOCNXCL`	Releases a previous exclusive access lock. See the comments above for details. The `ioctl` "arg" parameter is ignored.

Additional operation requests are also optionally available for the fast driver only, as shown in Table 8–3. To enable these operations in your program, you must set the preprocessor option `-DALTERA_AVALON_UART_USE_IOCTL`.

Table 8–3. Optional UART ioctl() Operations for the Fast Driver Only

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

Note to Table 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**.

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

Limitations

The HAL driver for the UART core does not support the endofpacket register. See "Register Map" for details.

Software Files

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

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

Legacy SDK Routines

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

Register Map

Programmers using the HAL API or the legacy SDK for the first-generation Nios processor never access the UART core directly via its registers. In general, the register map is only useful to programmers writing a device driver for the core.

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

Table 8–4 shows the register map for the UART core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 8–4. UART Core Register Map

Offset	Register Name	R/W	Description/Register Bits													
			15 . . .13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	`rxdata`	RO	*(1)*					*(2)*	*(2)*	Receive Data						
1	`txdata`	WO	*(1)*					*(2)*	*(2)*	Transmit Data						
2	`status` *(3)*	RW	*(1)*	eop	cts	dcts	(1)	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	`control`	RW	*(1)*	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	`divisor` *(4)*	RW	Baud Rate Divisor													
5	`endof-packet` *(4)*	RW	*(1)*					*(2)*	*(2)*	End-of-Packet Value						

Notes to Table 8–4:
(1) These bits are reserved. Reading returns an undefined value. Write zero.
(2) These bits may or may not exist, depending on the **Data Width** hardware option. If they do not exist, they read zero, and writing has no effect.
(3) Writing zero to the status register clears the dcts, e, toe, roe, brk, fe, and pe bits.
(4) This register may or may not exist, depending on hardware configuration options. If it does not exist, reading returns an undefined value and writing has no effect.

Some registers and bits are optional. These registers and bits exists in hardware only if it was enabled at system generation time. Optional registers and bits are noted below.

rxdata Register

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

txdata Register

Avalon-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 8–5.

Table 8–5. status Register Bits (Part 1 of 3)

Bit	Bit Name	Read/ Write/ Clear	Description
0 *(1)*	PE	RC	Parity error. A parity error occurs when the received parity bit has an unexpected (incorrect) logic level. The PE bit is set to 1 when the core receives a character with an incorrect parity bit. The PE bit stays set to 1 until it is explicitly cleared by a write to the status register. When the PE bit is set, reading from the rxdata register produces an undefined value. If the **Parity** hardware option is not enabled, no parity checking is performed and the PE bit always reads 0. See "Data Bits, Stop Bits, Parity" on page 8–6.
1	FE	RC	Framing error. A framing error occurs when the receiver fails to detect a correct stop bit. The FE bit is set to 1 when the core receives a character with an incorrect stop bit. The FE bit stays set to 1 until it is explicitly cleared by a write to the `status` register. When the FE bit is set, reading from the `rxdata` register produces an undefined value.
2	BRK	RC	Break detect. The receiver logic detects a break when the `RXD` pin is held low (logic 0) continuously for longer than a full-character time (data bits, plus start, stop, and parity bits). When a break is detected, the BRK bit is set to 1. The BRK bit stays set to 1 until it is explicitly cleared by a write to the `status` register.
3	ROE	RC	Receive overrun error. A receive-overrun error occurs when a newly received character is transferred into the `rxdata` holding register before the previous character is read (i.e., while the RRDY bit is 1). In this case, the ROE bit is set to 1, and the previous contents of `rxdata` are overwritten with the new character. The ROE bit stays set to 1 until it is explicitly cleared by a write to the `status` register.
4	TOE	RC	Transmit overrun error. A transmit-overrun error occurs when a new character is written to the `txdata` holding register before the previous character is transferred into the shift register (i.e., while the TRDY bit is 0). In this case the TOE bit is set to 1. The TOE bit stays set to 1 until it is explicitly cleared by a write to the `status` register.
5	TMT	R	Transmit empty. The TMT bit indicates the transmitter shift register's current state. When the shift register is in the process of shifting a character out the `TXD` pin, TMT is set to 0. When the shift register is idle (i.e., a character is not being transmitted) the TMT bit is 1. An Avalon-MM master peripheral can determine if a transmission is completed (and received at the other end of a serial link) by checking the TMT bit.

Table 8–5. status Register Bits (Part 2 of 3)

Bit	Bit Name	Read/ Write/ Clear	Description
6	TRDY	R	Transmit ready. The TRDY bit indicates the `txdata` holding register's current state. When the `txdata` register is empty, it is ready for a new character, and trdy is 1. When the `txdata` register is full, TRDY is 0. An Avalon-MM master peripheral must wait for TRDY to be 1 before writing new data to `txdata`.
7	RRDY	R	Receive character ready. The RRDY bit indicates the `rxdata` holding register's current state. When the `rxdata` register is empty, it is not ready to be read and rrdy is 0. When a newly received value is transferred into the `rxdata` register, RRDY is set to 1. Reading the `rxdata` register clears the RRDY bit to 0. An Avalon-MM master peripheral must wait for RRDY to equal 1 before reading the `rxdata` register.
8	E	RC	Exception. The E bit indicates that an exception condition occurred. The E bit is a logical-OR of the TOE, ROE, BRK, FE, and PE bits. The e bit and its corresponding interrupt-enable bit (IE) bit in the `control` register provide a convenient method to enable/disable IRQs for all error conditions. The E bit is set to 0 by a write operation to the status register.
10 *(1)*	DCTS	RC	Change in clear to send (CTS) signal. The DCTS bit is set to 1 whenever a logic-level transition is detected on the CTS_N input port (sampled synchronously to the Avalon-MM clock). This bit is set by both falling and rising transitions on CTS_N. The DCTS bit stays set to 1 until it is explicitly cleared by a write to the `status` register. If the **Flow Control** hardware option is not enabled, the DCTS bit always reads 0. See "Flow Control" on page 8–6.
11 *(1)*	CTS	R	Clear-to-send (CTS) signal. The CTS bit reflects the CTS_N input's instantaneous state (sampled synchronously to the Avalon-MM clock). Because the CTS_N input is logic negative, the CTS bit is 1 when a 0 logic-level is applied to the CTS_N input. The CTS_N input has no effect on the transmit or receive processes. The only visible effect of the CTS_N input is the state of the CTS and DCTS bits, and an IRQ that can be generated when the control register's idcts bit is enabled. If the **Flow Control** hardware option is not enabled, the CTS bit always reads 0. See "Flow Control" on page 8–6.

Table 8–5. status Register Bits (Part 3 of 3)

Bit	Bit Name	Read/ Write/ Clear	Description
12 *(1)*	EOP	R	End of packet encountered. The EOP bit is set to 1 by one of the following events: ● An EOP character is written to `txdata` ● An EOP character is read from `rxdata` The EOP character is determined by the contents of the `endofpacket` register. The EOP bit stays set to 1 until it is explicitly cleared by a write to the `status` register. If the **Include End-of-Packet Register** hardware option is not enabled, the EOP bit always reads 0. See "Avalon-MM Transfers With Flow Control (DMA)" on page 8–7.

Note to Table 8–5:
(1) This bit is optional and may not exist in hardware.

control Register

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

Each bit in the `control` register enables an IRQ for a corresponding bit in the `status` register. When both a status bit and its corresponding interrupt-enable bit are 1, the core generates an IRQ. For example, the pe bit is bit 0 of the `status` register, and the ipe bit is bit 0 of the `control` register. An interrupt request is generated when both pe and ipe equal 1.

The control register bits are shown in Table 8–6.

Table 8–6. control Register Bits (Part 1 of 2)

Bit	Bit Name	Read/ Write	Description
0	IPE	RW	Enable interrupt for a parity error.
1	IFE	RW	Enable interrupt for a framing error.
2	IBRK	RW	Enable interrupt for a break detect.
3	IROE	RW	Enable interrupt for a receiver overrun error.
4	ITOE	RW	Enable interrupt for a transmitter overrun error.
5	ITMT	RW	Enable interrupt for a transmitter shift register empty.

Table 8–6. control Register Bits (Part 2 of 2)

Bit	Bit Name	Read/ Write	Description
6	ITRDY	RW	Enable interrupt for a transmission ready.
7	IRRDY	RW	Enable interrupt for a read ready.
8	IE	RW	Enable interrupt for an exception.
9	TRBK	RW	Transmit break. The TRBK bit allows an Avalon-MM master peripheral to transmit a break character over the `TXD` output. The `TXD` signal is forced to 0 when the TRBK bit is set to 1. The TRBK bit overrides any logic level that the transmitter logic would otherwise drive on the `TXD` output. The TRBK bit interferes with any transmission in process. The Avalon-MM master peripheral must set the TRBK bit back to 0 after an appropriate break period elapses.
10	IDCTS	RW	Enable interrupt for a change in CTS signal.
11 *(1)*	RTS	RW	Request to send (RTS) signal. The RTS bit directly feeds the RTS_N output. An Avalon-MM master peripheral can write the RTS bit at any time. The value of the RTS bit only affects the RTS_N output; it has no effect on the transmitter or receiver logic. Because the RTS_N output is logic negative, when the RTS bit is 1, a low logic-level (0) is driven on the RTS_N output. If the **Flow Control** hardware option is not enabled, the RTS bit always reads 0, and writing has no effect. See "Flow Control" on page 8–6.
12	IEOP	RW	Enable interrupt for end-of-packet condition.

Note to Table 8–6:
(1) This bit is optional and may not exist in hardware.

divisor Register (Optional)

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

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

The `divisor` register is an optional hardware feature. If the **Baud Rate Can Be Changed By Software** hardware option is not enabled, then the `divisor` register does not exist. In this case, writing `divisor` has no effect, and reading `divisor` returns an undefined value. For more information see "Baud Rate Options" on page 8–5.

endofpacket Register (Optional)

The value in the `endofpacket` register determines the end-of-packet character for variable-length DMA transactions. After reset, the default value is zero, which is the ASCII null character (\0). For more information, see Table 8–5 on page 8–16 for the description for the eop bit.

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

Interrupt Behavior

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

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

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

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

Referenced Documents

This chapter references the following documents:

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

Document Revision History

Table 8–7 shows the revision history for this chapter.

Table 8–7. Document Revision History

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