Using Terminals with DE-Series Boards

For Quartus Prime 16.1

1 Introduction

This tutorial provides an introduction to using terminals to communicate with programs compiled for Nios II and ARM processors on Intel's DE-series FPGA boards. The tutorial demonstrates how to write C and assembly code for these processors to send and receive characters to and from a host PC. On the host PC side, the tutorial describes how to use the Intel FPGA Monitor Program's terminal window, as well as third-party terminal programs such as Putty, to send and receive characters to and from the FPGA board.

The reader is expected to have a basic understanding of C and assembly languages, and to be familiar with the Intel FPGA Monitor Program software.

Contents:

- Introduction to terminals
- Using the JTAG UART terminal
- Using the RS232 UART terminal
- Using the USB UART terminal
- Using the ARM Semihosting terminal

2 Background

A terminal is a program that receives and displays text data; it also allows a user to input text data for transmission to other programs. Most programmers will be familiar with the Linux console (a kind of terminal), where a program can call the printf and scanf functions to display text to, and read text from, the terminal. In this case, the program and the terminal both run on the same system, and the communication between the two is facilitated by the Linux operating system.

Sometimes, it is necessary to use a terminal to communicate with a remote program - one that is running on a different system than the terminal. For example, when developing a program for a Nios II or ARM processor running on an Intel FPGA board, it is convenient to use a terminal on the host PC to communicate with the program running on the board. Because the terminal and the remote program run on separate systems, some sort of communication link between the two systems is required for data transmission. The DE-series FPGA boards provide a number of ways to establish a communication link with the host PC: JTAG UART, RS232 UART, USB UART, and Semihosting. This tutorial describes how to write remote program code for Nios II and ARM processors to send and receive characters through these communication links. As well, it describes how to launch a terminal on the host PC to connect to these links.

3 Using Terminals

This section describes the use of the JTAG UART, RS232 UART, USB UART, and Semihosting terminals. Because each terminal type requires a particular communication link, some boards may not support certain terminals. For example, the RS232 UART terminal uses a RS232 serial cable to transmit characters. Only the DE0 and DE2-115 boards contain RS232 ports and therefore support this terminal type. Table 1 provides a full list of the terminals supported by each DE-series board.

Table 1. Supported Terminals by Board

	JTAG UART	RS232 UART	USB UART	Semihosting
DE0-CV	Yes			
DE0-Nano	Yes			
DE0-Nano-SoC	Yes		Yes	ARM Only
DE1-SoC	Yes		Yes	ARM Only
DE2-115	Yes	Yes		

3.1 The JTAG UART Terminal

The JTAG UART is an IP core that facilitates serial UART communication between the system on the FPGA board and the host PC, through a JTAG cable such as the Intel USB Blaster I/II. While this communication link can be used to transmit any data, it is in practice commonly used to transmit character data. Figure 1 shows the arrangement of the components used to facilitate this communication link. These components are described in the following paragraphs.

On the FPGA board, a program running on the Nios II or ARM processor uses the JTAG UART core's memory-mapped register interface to send and receive character data to and from the PC. When the program writes characters

to the JTAG UART's register interface, the JTAG UART sends these characters through the USB Blaster I/II cable to the host PC. When the JTAG UART receives characters from the host PC via the USB Blaster I/II cable, it stores these characters in a FIFO queue, which can be read by the program through the JTAG UART's memory mapped registers.

On the host PC, the Intel FPGA Monitor Program's built-in *Terminal window* can be used to send and receive character data with the JTAG UART. To do this, the Intel FPGA Monitor Program must be configured to communicate with the JTAG UART by selecting it in the *Terminal device* drop-down menu in the *System Settings* window. Now, upon starting a debugging session with the FPGA board, the Monitor Program will poll the JTAG server for any characters coming from the JTAG UART and display these characters in its *Terminal window*. As well, characters inputted to the *Terminal window* by the user are sent through the JTAG server to the JTAG UART. The JTAG Server is a program included with Quartus Prime software that allows programs like the Intel FPGA Monitor Program to send and receive data through a USB Blaster I/II cable.

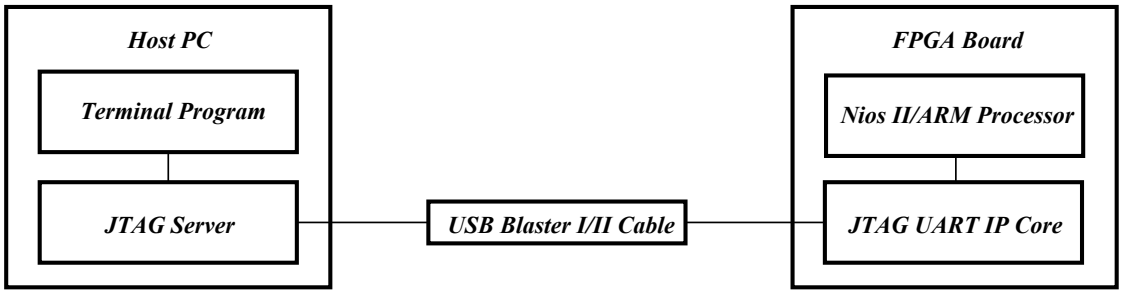

Figure 1. A JTAG UART communication link between the host PC and the FPGA board.

3.1.1 Using the JTAG UART Register Interface

The JTAG UART includes a transmit (TX) FIFO queue that stores data waiting to be transmitted to the host computer. The size of this FIFO queue is 64 characters by default, and can be configured when the core is instantiated in Qsys. Character data is loaded into this FIFO queue by performing a write to bits 7−0 of the *Data* register in Figure 2. Note that writing into this register has no effect on received data. The amount of space, *WSPACE*, currently available in the TX FIFO queue is provided in bits 31−16 of the *Control* register. If the TX FIFO queue is full, then any characters written to the *Data* register will be lost.

Offset	31 · · · 16	15	14 · · · 11	10	9	8	7 · · · 1	0	
0x0	RAVAIL	RVALID	Unused				DATA		Data register
0x4	WSPACE	Unused		AC	WI	RI		WE	RE	Control register

Figure 2. JTAG UART registers.

When character data from the host computer is received by the JTAG UART it is stored in a receive (RX) FIFO queue. The size of this FIFO queue is 64 characters by default, and can be configured when the core is instantiated in Qsys. The number of characters currently stored in this FIFO queue is indicated in the field *RAVAIL*, which are

bits 31−16 of the *Data* register. If the RX FIFO queue overflows, then additional data is lost. When data is present in the RX FIFO queue, then the value of *RAVAIL* will be greater than 0 and the value of bit 15, *RVALID*, will be 1. Reading the character at the head of the FIFO queue, which is provided in bits 7−0, decrements the value of *RAVAIL* by one and returns this decremented value as part of the read operation. If no data is present in the RX FIFO queue, then *RVALID* will be set to 0 and the data in bits 7 − 0 is invalid.

Bit 10 in the *Control* register, called *AC*, has the value 1 if the JTAG UART has been accessed by the host computer. This bit can be used to check if a working connection to the host computer has been established. The *AC* bit can be cleared to 0 by writing a 1 into it.

The *WE* and *RE* bits of the *Control* register can be set to 1 to enable write and read interrupts, respectively. The *WI* and *RI* bits have the value 1 if a write or read interrupt is pending, respectively. This tutorial does not describe the use of interrupts with the JTAG UART IP Core.

3.1.2 Using the JTAG UART with C Code

Figure 3 shows two C functions for reading and writing the JTAG UART, respectively. The *get_char* function attempts to read a previously unread character from the JTAG UART, and returns that character if successful. If a previously unread character is not available in the RX FIFO queue a null character '\0' is returned instead.

The *put_char* function attempts to write a character to the JTAG UART. It succeeds if there is space available in the TX FIFO queue of the JTAG UART which it determines by reading bits 31−16 of the *Control* register. If there is no space, the function does not write the character to the JTAG UART.

An example C program that demonstrates the use of the JTAG UART is made available as part of the Intel FPGA Monitor Program. The example can be found under the heading *sample programs*, and is identified by the name *JTAG UART*.

3.1.3 Using the JTAG UART with Nios II Assembly Code

Figure 4 shows two Nios II assembly-language subroutines for reading and writing the JTAG UART. The *GET_CHAR* subroutine reads a character from the JTAG UART, and returns that character in register r2. If the JTAG UART's RX FIFO queue is empty, it returns the null character '\0'.

The *PUT_CHAR* subroutine attempts to write a character to the JTAG UART. It succeeds if there is space available in the TX FIFO queue of the JTAG UART which it determines by reading bits 31−16 of the *Control register*. If there is no space, the subroutine skips the *stwio* instruction, thus not writing the character to the JTAG UART.

An example Nios II assembly-language program that demonstrates the use of the JTAG UART is made available as part of the Intel FPGA Monitor Program. The example can be found under the heading *sample programs*, and is identified by the name *JTAG UART*.

```
/*****************************************************************************
 * Subroutine to read a character from the JTAG UART
 * Returns \0 if no character, otherwise returns the character
*****************************************************************************/
char get_char( void )
{
    volatile int * JTAG_UART_ptr = (int *) 0xFF201000; // JTAG UART address
    int data;

    data = *(JTAG_UART_ptr);            // read the JTAG_UART data register
    if (data & 0x00008000)              // check RVALID to see if there is new data
        return ((char) data & 0xFF);
    else
        return ('\0');
}
/*****************************************************************************
 * Subroutine to send a character to the JTAG UART
*****************************************************************************/
void put_char( char c )
{
    volatile int * JTAG_UART_ptr = (int *) 0xFF201000; // JTAG UART address
    int control;
    control = *(JTAG_UART_ptr + 1);     // read the JTAG_UART control register
    if (control & 0xFFFF0000)           // if space, write character, else ignore
        *(JTAG_UART_ptr) = c;
}
```

Figure 3. C-language functions that use the JTAG UART.

```
/*****************************************************************************
 * Subroutine to read a character from the JTAG UART
 *       r4  = JTAG UART base address
 * Returns the character in r2. Returns '\0' if no new character in RX FIFO.
*****************************************************************************/
            .global     GET_CHAR
GET_CHAR:
            /* save any modified registers */
            subi        sp, sp, 8               /* reserve space on the stack */
            stw         r5, 0(sp)               /* save register */
            ldwio       r2, 0(r4)               /* read the JTAG UART Data register */
            andi        r5, r2, 0x8000          /* check if there is new data */
            bne         r5, r0, RETURN_CHAR
            mov         r2, r0                  /* if no new data, return '\0' */
RETURN_CHAR:
            andi        r5, r2, 0x00ff          /* the data is in the least significant byte */
            mov         r2, r5                  /* set r2 with the return value */
            /* restore registers */
            ldw         r5, 0(sp)
            addi        sp, sp, 8

            ret

/*****************************************************************************
 * Subroutine to send a character to the JTAG UART.
 *       r4  = JTAG UART base address
 *       r5  = character to send
*****************************************************************************/
            .global     PUT_CHAR
PUT_CHAR:
            /* save any modified registers */
            subi        sp, sp, 4               /* reserve space on the stack */
            stw         r6, 0(sp)               /* save register */

            ldwio       r6, 4(r4)               /* read the JTAG UART Control register */
            andhi       r6, r6, 0xffff          /* check for write space */
            beq         r6, r0, END_PUT         /* if no space, ignore the character */
            stwio       r5, 0(r4)               /* send the character */
END_PUT:
            /* restore registers */
            ldw         r6, 0(sp)
            addi        sp, sp, 4

            ret
```

Figure 4. Nios II assembly-language subroutines that use the JTAG UART.

3.1.4 Using the JTAG UART with ARM Assembly Code

Figure 5 shows two ARM assembly-language subroutines for reading and writing the JTAG UART, respectively. The *GET_CHAR* subroutine reads a character from the JTAG UART, and returns that character in register R0. If the JTAG UART's RX FIFO queue is empty, it returns the null character '\0'.

The *PUT_CHAR* subroutine attempts to write a character to the JTAG UART. It succeeds if there is space available in the TX FIFO queue of the JTAG UART which it determines by reading bits 31−16 of the *Control* register. If there is no space, the subroutine skips the *STR* instruction, thus not writing the character to the JTAG UART.

An example ARM assembly-language program that demonstrates the use of the JTAG UART is made available as part of the Intel FPGA Monitor Program. The example can be found under the heading *sample programs*, and is identified by the name *JTAG UART*.

```
/********************************************************************************
 * Subroutine to get a character from the JTAG UART
 * R1 = JTAG UART base address
 * Returns the character read in R0
********************************************************************************/
                .global     GET_CHAR
GET_CHAR:
                LDR         R0, [R1]                // read the JTAG UART data register
                ANDS        R2, R0, #0x8000         // check if there is new data
                BNE         RETURN_CHAR
                MOV         R0, #0                  // if no data, return '\0'
RETURN_CHAR:
                AND         R0, R0, #0x00FF         // return the character
                BX          LR

/********************************************************************************
 * Subroutine to send a character to the JTAG UART
 * R0 = character to send
 * R1 = JTAG UART base address
********************************************************************************/
                .global     PUT_CHAR
PUT_CHAR:
                LDR         R2, [R1, #4]            // read the JTAG UART control register
                LDR         R3, =0xFFFF0000
                ANDS        R2, R2, R3              // check for write space
                BEQ         END_PUT                 // if no space, ignore the character
                STR         R0, [R1]                // send the character
END_PUT:
                BX          LR
```

Figure 5. ARM assembly-language subroutines that use the JTAG UART.

3.2 The RS232 UART Terminal

The RS232 UART is an IP core that facilitates serial UART communication between the system on the FPGA board and the host PC, through an RS232 cable. Figure 6 shows the arrangement of the components used to facilitate this communication link. While this serial communication link can be used to transmit any arbitrary data, it is in practice commonly used to transmit character data. On the FPGA board, a program running on the Nios II or ARM processor can use the RS232 core's memory-mapped register interface shown in Figure 7 to send and receive character data to and from the host PC.

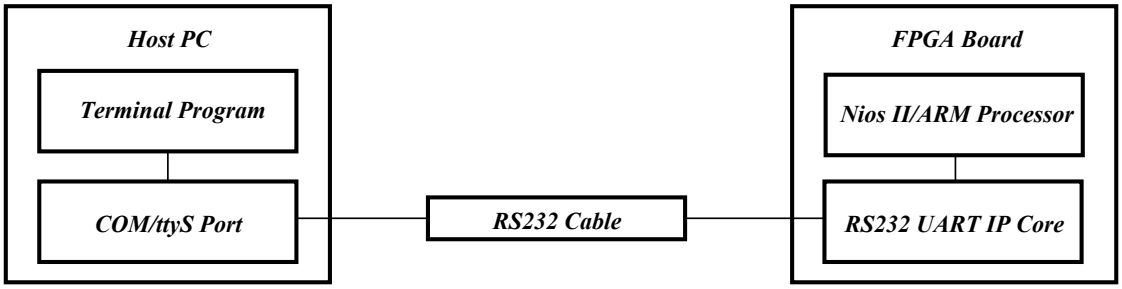

Figure 6. An RS232 UART communication link between the host PC and the FPGA board.

When the RS232 UART is connected to a host PC via an RS232 cable, the PC will recognize it as a serial communication device. On a Windows PC, it will appear as a *COM* port (in *Device Manager*). On a Linux PC, it will appear as a *ttyS* character device (in */dev/*). There exist many terminal programs that can be used to communicate with a serial communication device, and documentation for them can be found online. Section 3.2.5 of this tutorial demonstrates the use of a popular and free terminal called *Putty* to communicate with the RS232 UART.

3.2.1 Using the RS232 UART Register Interface

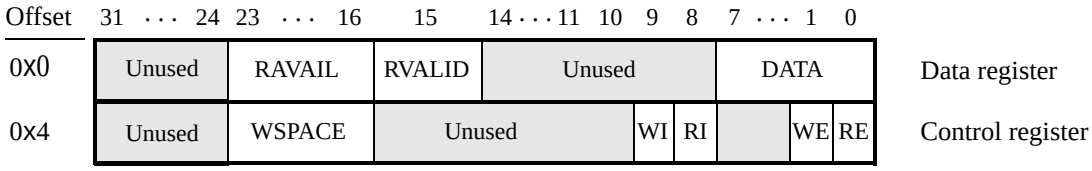

Figure 7. RS232 UART registers.

The RS232 UART includes a transmit (TX) FIFO queue that stores data waiting to be transmitted to the host computer. Character data is loaded into this FIFO queue by performing a write to bits 7−0 of the *Data* register in Figure 7. Note that writing into this register has no effect on received data. The amount of space, *WSPACE*, currently available in the TX FIFO queue is provided in bits 23−16 of the *Control* register. If the TX FIFO queue is full, then any characters written to the *Data* register will be lost.

When character data from the host computer is received by the RS232 UART it is stored in a receive (RX) FIFO queue. The number of characters currently stored in this FIFO queue is indicated in the field *RAVAIL*, which are bits 23−16 of the *Data* register. If the RX FIFO queue overflows, then additional data is lost. When data is present in the

RX FIFO queue, then the value of *RAVAIL* will be greater than 0 and the value of bit 15, *RVALID*, will be 1. Reading the character at the head of the FIFO queue, which is provided in bits $7-0$, decrements the value of *RAVAIL* by one and returns this decremented value as part of the read operation. If no data is present in the RX FIFO queue, then *RVALID* will be set to 0 and the data in bits $7-0$ is invalid.

The *WE* and *RE* bits of the *Control* register can be set to 1 to enable write and read interrupts, respectively. The *WI* and *RI* bits have the value 1 if a write or read interrupt is pending, respectively. This tutorial does not describe the use of interrupts with the RS232 UART IP Core.

3.2.2 Using the RS232 UART with C Code

Figure 8 shows two C functions for reading and writing the RS232 UART, respectively. The *get_char* function attempts to read a previously unread character from the RS232 UART, and returns that character if successful. If a previously unread character is not available in the RX FIFO queue a null character '\0' is returned instead.

The *put_char* function attempts to write a character to the RS232 UART. It succeeds if there is space available in the TX FIFO queue of the RS232 UART which it determines by reading bits 23−16 of the *Control* register. If there is no space, the function does not write the character to the RS232 UART.

3.2.3 Using the RS232 UART with Nios II Assembly Code

Figure 9 shows two Nios II assembly-language subroutines for reading and writing the RS232 UART, respectively. The *GET_CHAR* subroutine reads a character from the RS232 UART, and returns that character in register r2. If the RS232 UART's RX FIFO queue is empty, it returns the null character '\0'.

The *PUT_CHAR* subroutine attempts to write a character to the RS232 UART. It succeeds if there is space available in the TX FIFO queue of the RS232 UART which it determines by reading bits 23−16 of the *Control register*. If there is no space, the subroutine skips the *stwio* instruction, thus not writing the character to the RS232 UART.

```
/*****************************************************************************
 * Subroutine to read a character from the RS232 UART
 * Returns \0 if no character, otherwise returns the character
*****************************************************************************/
char get_char( void )
{
    volatile int * RS232_UART_ptr = (int *) 0xFF201000; // RS232 UART address
    int data;

    data = *(RS232_UART_ptr);           // read the RS232_UART data register
    if (data & 0x00008000)              // check RVALID to see if there is new data
        return ((char) data & 0xFF);
    else
        return ('\0');
}
/*****************************************************************************
 * Subroutine to send a character to the RS232 UART
*****************************************************************************/
void put_char( char c )
{
    volatile int * RS232_UART_ptr = (int *) 0xFF201000; // RS232 UART address
    int control;
    control = *(RS232_UART_ptr + 1);   // read the RS232_UART control register
    if (control & 0x00FF0000)          // if space, write character, else ignore
        *(RS232_UART_ptr) = c;
}
```

Figure 8. C-language functions that use the RS232 UART.

```
/*****************************************************************************
 * Subroutine to read a character from the RS232 UART
 *       r4  = RS232 UART base address
 * Returns the character in r2. Returns '\0' if no new character in RX FIFO queue.
*****************************************************************************/
            .global     GET_CHAR
GET_CHAR:
            /* save any modified registers */
            subi        sp, sp, 8               /* reserve space on the stack */
            stw         r5, 0(sp)               /* save register */
            ldwio       r2, 0(r4)               /* read the RS232 UART Data register */
            andi        r5, r2, 0x8000          /* check if there is new data */
            bne         r5, r0, RETURN_CHAR
            mov         r2, r0                  /* if no new data, return '\0' */
RETURN_CHAR:
            andi        r5, r2, 0x00ff          /* the data is in the least significant byte */
            mov         r2, r5                  /* set r2 with the return value */
            /* restore registers */
            ldw         r5, 0(sp)
            addi        sp, sp, 8

            ret

/*****************************************************************************
 * Subroutine to send a character to the RS232 UART.
 *       r4  = RS232 UART base address
 *       r5  = character to send
*****************************************************************************/
            .global     PUT_CHAR
PUT_CHAR:
            /* save any modified registers */
            subi        sp, sp, 4               /* reserve space on the stack */
            stw         r6, 0(sp)               /* save register */

            ldwio       r6, 4(r4)               /* read the RS232 UART Control register */
            andhi       r6, r6, 0x00ff          /* check for write space */
            beq         r6, r0, END_PUT         /* if no space, ignore the character */
            stwio       r5, 0(r4)               /* send the character */
END_PUT:
            /* restore registers */
            ldw         r6, 0(sp)
            addi        sp, sp, 4

            ret
```

Figure 9. Nios II assembly-language subroutines that use the RS232 UART.

3.2.4 Using the RS232 UART with ARM Assembly Code

Figure 10 shows two ARM assembly code subroutines for reading and writing the RS232 UART, respectively. The *GET_CHAR* subroutine reads a character from the RS232 UART, and returns that character in register R0. If the RS232 UART's RX FIFO queue is empty, it returns the null character '\0'.

The *PUT_CHAR* subroutine attempts to write a character to the RS232 UART. It succeeds if there is space available in the TX FIFO queue of the RS232 UART which it determines by reading bits 23−16 of the *Control register*. If there is no space, the subroutine skips the *STR* instruction, thus not writing the character to the RS232 UART.

```
/*****************************************************************************
 * Subroutine to get a character from the RS232 UART
 * R1 = RS232 UART base address
 * Returns the character read in R0
*****************************************************************************/
                .global     GET_CHAR
GET_CHAR:
                LDR         R0, [R1]                // read the RS232 UART data register
                ANDS        R2, R0, #0x8000         // check if there is new data
                BNE         RETURN_CHAR
                MOV         R0, #0                  // if no data, return '\0'
RETURN_CHAR:
                AND         R0, R0, #0x00FF         // return the character
                BX          LR

/*****************************************************************************
 * Subroutine to send a character to the RS232 UART
 * R0 = character to send
 * R1 = RS232 UART base address
*****************************************************************************/
                .global     PUT_CHAR
PUT_CHAR:
                LDR         R2, [R1, #4]            // read the RS232 UART control register
                LDR         R3, =0x00FF0000
                ANDS        R2, R2, R3              // check for write space
                BEQ         END_PUT                 // if no space, ignore the character
                STR         R0, [R1]                // send the character
END_PUT:
                BX          LR
```

Figure 10. ARM assembly language subroutines that use the RS232 UART.

3.2.5 Connecting to the RS232 UART from the Host PC Using Putty

Putty is a popular free-to-use terminal program that can be used to communicate with serial character devices attached to the PC. This section describes how to use Putty to connect to the RS232 UART on a Windows host PC.

On a Windows PC, serial communication devices such as the RS232 UART are recognized by the PC as COM ports. As there can be multiple COM ports connected to the PC, each COM port is assigned a unique identifying number. The number assigned to the RS232 UART can be determined by viewing the list of COM ports in *Device Manager*. Figure 11 shows the *Device Manager's* list of available COM ports on one particular PC. Here, there was only one COM port (the RS232 UART) connected which had been assigned the number 5 (COM5). In the case where there are many COM ports available, the number assigned to the RS232 UART can be determined by disconnecting and reconnecting the cable to see which COM port disappears then reappears in the list.

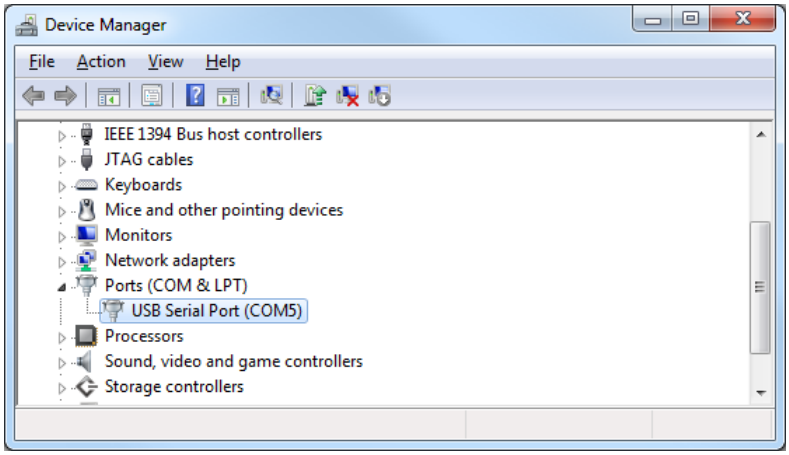

Figure 11. Determining the COM number assigned to the RS232 UART in Device Manager.

Once the COM port corresponding to the RS232 UART is determined, Putty can be configured to connect to it. Figure 12 shows the main window of Putty. In this window, the *Serial* connection type must be chosen, and the COM port must be entered in the *Serial line* field, as shown in the figure.

Some additional details about the RS232 UART must be entered by selecting the *Serial* panel in the *Category* box on the left side of the window. The *Serial* panel is shown in Figure 13. These settings must be configured to match the settings that were chosen when instantiating the RS232 UART IP core. Figure 14 shows the parameters that are set when instantiating the core in *Qsys*, including the *Baud rate*, *Parity*, *Data bits*, and *Stop bits*. The *Flow control* option should be set to *None* as the RS232 UART IP core does not support any flow control protocol.

The *Speed* option refers to the *baud rate* of the the RS232 UART, which is the number of bits per second that the RS232 UART will send, and is expecting to receive from the host PC. The *Data bits* option refers to how many bits of data will be transmitted in each packet. Eight data bits are typically used for transmitting ASCII characters. The *Stop bits* refer to how many bits follow the data bits to indicate that the packet is ending. The *Parity* option refers to the error-checking scheme to use to determine if there is data corruption in incoming packets. The *Flow control* option refers to the handshaking protocol by which either end of the connection can request to pause and resume transmission of data.

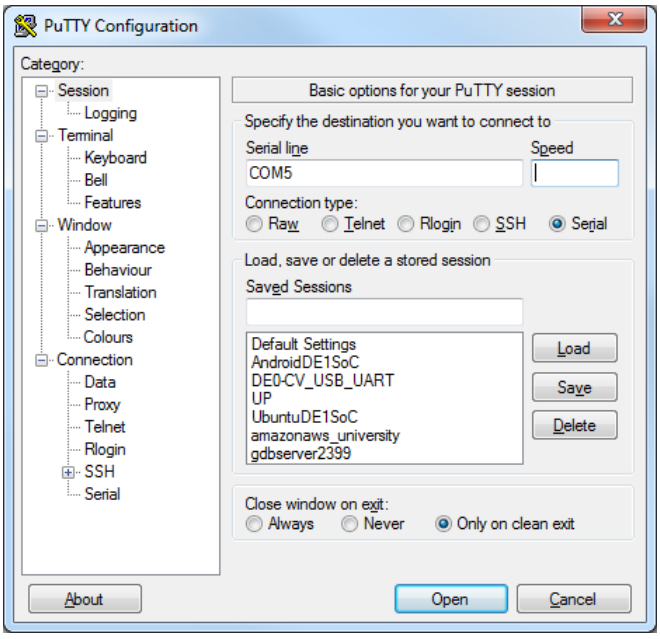

Figure 12. Putty's main window.

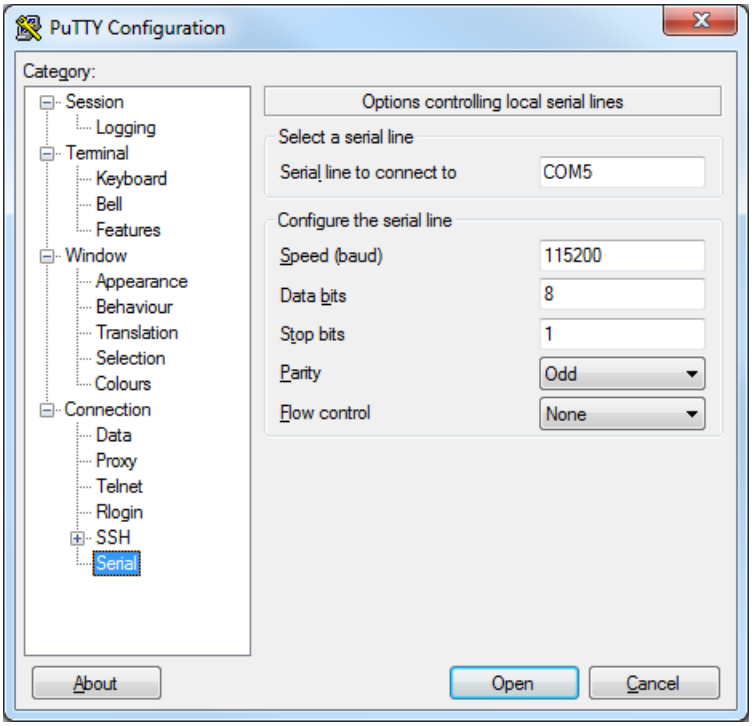

Figure 13. Putty's configuration window for serial communication settings.

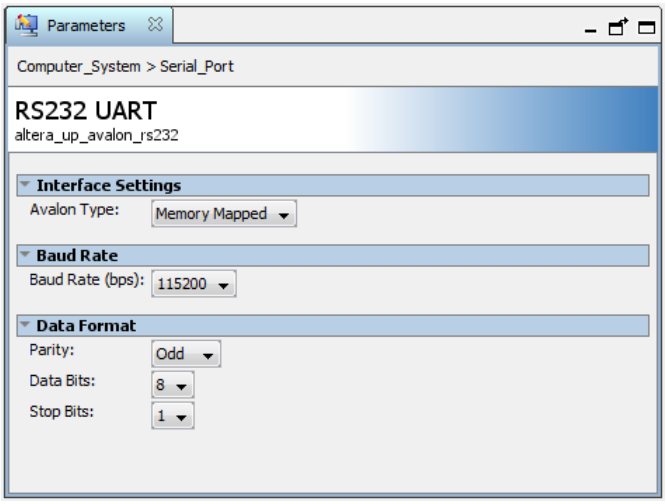

Figure 14. RS232 UART settings in Qsys Wizard.

Once all of the serial line settings have been entered, the *Open* button can be pressed to start the terminal. Once opened, a terminal will appear as shown in Figure 15. This terminal will now display text coming from the RS232 UART, and send any user inputs to the RS232 UART.

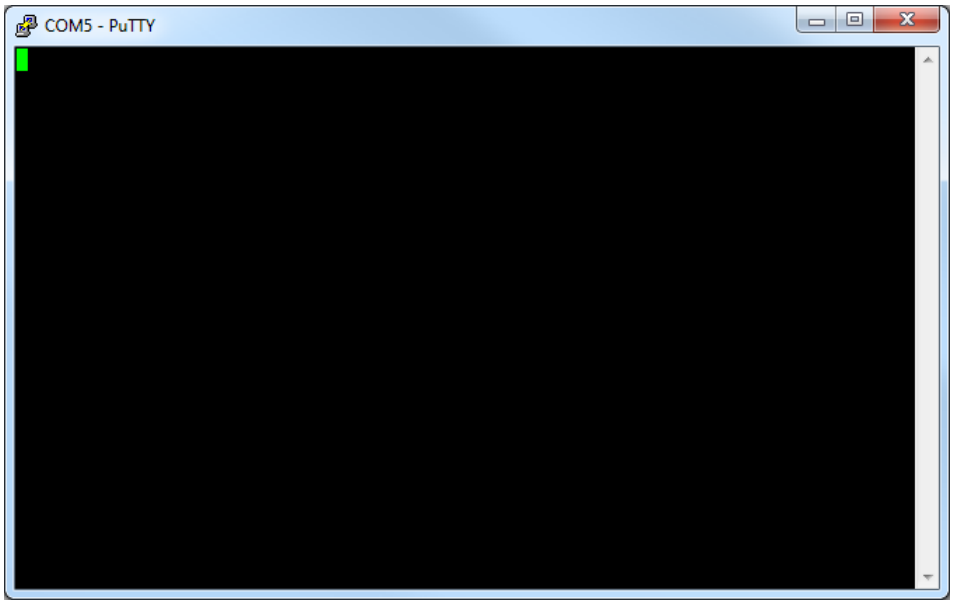

Figure 15. Putty terminal ready to communicate with RS232 UART.

3.3 The USB UART Terminal

Serial UART communication to facilitate terminal communication in the past was commonly done using an RS232 cable. As modern computers have adopted the USB cable as the preferred connection medium, few computers now have RS232 ports. As a result, USB-to-UART chips are often used to allow systems (such as a system on a DE-series board) to perform serial UART communication over USB cable with a host PC. The DE1-SoC board contains the *FTDI FT232R* USB-to-UART chip and is capable of this communication link. Figure 16 shows the arrangement of the components that are used to facilitate this communication link.

On the FPGA board, the USB-to-UART chip exposes an RS232 interface, which comprises an RX (receive) pin, and a TX (transmit) pin. The RS232 UART IP core can be instantiated to communicate through these pins. A program running on a Nios II or ARM processor on the board can then use this IP core's register interface, described in section 3.2.1, to send and receive characters to the host PC. C code that uses this interface is shown in section 3.2.2. Nios II assembly-language code that uses this interface is shown in section 3.2.3. ARM assembly-language code that uses this interface is shown in section 3.2.4.

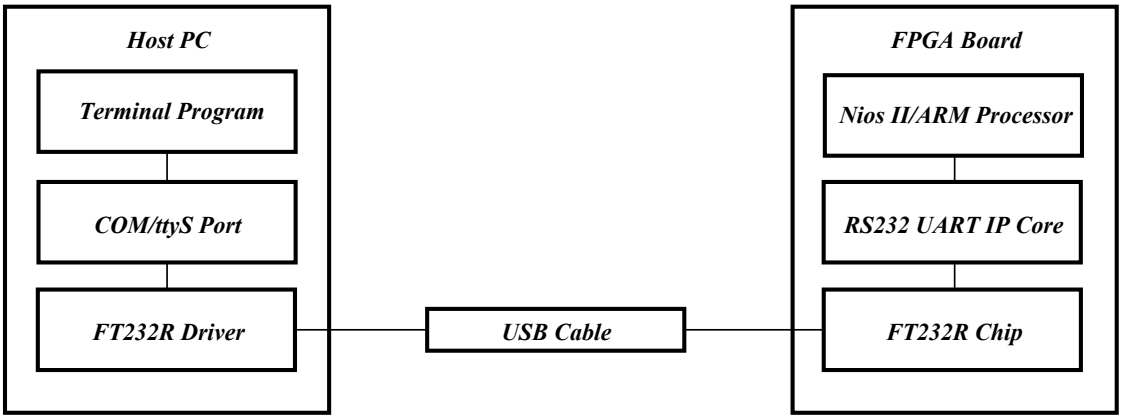

Figure 16. A USB UART communication link between the host PC and the FPGA board.

On the host PC side, the driver for the *FTDI FT232R* chip detects that the chip is connected via a USB cable, and exposes it as a serial communication device. On a Windows PC, it will appear as a *COM* port (in *Device Manager*). On a Linux PC, it will appear as a *ttyS* character device (in */dev/*). A terminal program can connect to the exposed COM/ttyS interface to send and receive characters with the system on the FPGA board. There exist many terminal programs that can be used, and documentation for them can be found online. Section 3.2.5 of this tutorial demonstrates the use of a popular and free terminal program called *Putty*.

3.4 The ARM Semihosting Terminal

When a debugger such as the Intel FPGA Monitor Program starts a debugging session with a processor, it establishes a communication link to the processor. It is through this link that the debugger performs operations such as single-stepping and setting breakpoints. Semihosting is a mechanism that leverages this connection to allow a program running on an ARM processor to request services from the debugger. Among other possible services, a program can request to read and write files. When a program writes characters to a special file named *stdout*, the Intel

FPGA Monitor Program displays those characters in its Semihosting terminal. When a user enters characters to the Semihosting terminal, these characters are stored in a special file named *stdin* which can be read by the program. Note that *stdout* and *stdin* are not actually files, and are not stored in persistent storage. They are temporary buffers allocated inside the Intel FPGA Monitor Program, and are lost when the Intel FPGA Monitor Program closes.

To use the Semihosting Terminal in the Intel FPGA Monitor Program, users must select it under the *Terminal device* drop-down menu in the *System Settings* window. Once selected, the Intel FPGA Monitor Program's *Terminal window* will function as a Semihosting terminal upon starting a debugging session with the ARM processor.

The following subsection describes how to write ARM programs in C to use the Semihosting interface to read and write *stdin* and *stdout*. It is not recommended to use the Semihosting interface in assembly-language code. When writing ARM assembly-language programs, the JTAG UART or RS232 UART should be used instead.

3.4.1 Using Semihosting with C Code

The ARM C compiler that is used in the Intel FPGA Monitor Program uses special C libraries that have been modified to use the Semihosting interface. Standard I/O functions such as printf, scanf, puts, and fprintf that are built into these libraries will automatically use Semihosting services to perform their respective tasks. This provides a Linux-like way to perform terminal communication, with no need to manually interface with the underlying communication link. Figure 17 shows an example C program that uses these functions to print strings to the terminal and accept input from the user.

```
/*****************************************************************************
 * C Program that asks the user for their name then prints it in three different ways.
*****************************************************************************/
#include <stdio.h>

int main(void){
    char name[64];

    // Get user's name
    printf("Hello, what is your name?\n");
    scanf("%s",name);

    printf("Your name is:\n");

    // Print the user's name in 3 different ways
    printf("%s\n",name);
    puts(name);
    fprintf(stdout, "%s\n", name);

    return 0;
}
```

Figure 17. ARM C code that uses the Semihosting terminal.

4 Conclusion

This tutorial demonstrated the use of the different terminals available for communicating with programs running on DE-series boards.