

29/11/2021, 3/12/2021 Laboratory

Introduction

In this laboratory session you will learn how to configure and manage a processor peripheral of the Nios II processor. The peripheral implements the UART protocol for both receiving and transmitting data, and it will interface with a terminal emulator program on a PC. Use a USB-mini cable to make the connection between the DE1-SoC board and the PC. The FDTI232R integrated circuit on the DE1-SoC board transforms the UART signals into a USB pipe, and a USB driver on the PC transforms the USB pipe into a virtual COM (serial, i.e. COM3) port for the terminal emulator. To know the virtual COM port number you can use the Device Manager in Windows. Figure 1 shows a block diagram of the overall system.

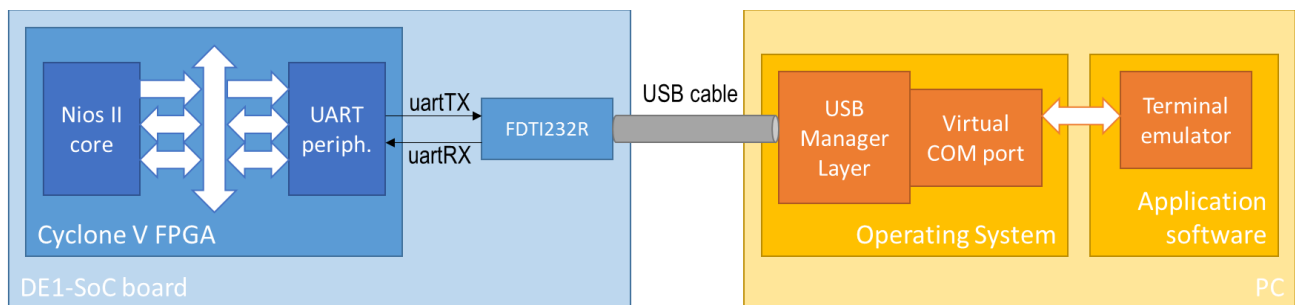


Figure 1: System block diagram for testing the UART communication protocol

The description of the UART peripheral is available in Chapter 11 of the “Embedded Peripheral IP User Guide” from Intel (starting at page 128 in the PDF file). The base address assigned to the UART peripheral in Platform Designer is (as you can see in the file system.h in the software project):

```
#define UART_0_BASE 0x8001060
```

The UART peripheral configuration at generation time in Platform Designer is the following:

- Baud Rate = 115200 (variable)
- Data Bits = 8
- Stop Bits = 1
- Parity = Even

With these parameters, all registers in the peripheral are available, except the End-of-Packet Register that is not activated. The register map is the following:

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	Reserved						Receive Data							
1	txdata	WO	Reserved						Transmit Data							
2	status	RW	Reserved	eop	cts	dcts		e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	control	RW	Reserved	ieop	rts	idcts	tbrk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	divisor	RW	Baud Rate Divisor													

Please read the description of the various bits of the STATUS and CONTROL registers in the User Guide provided by Intel. Focus in particular on PE (Parity Error), ROE (Receive Overrun Error), TOE (Transmit Overrun Error), TMT (Transmit Empty), TRDY (Transmit Ready), RRDY (Receive Ready), E (Exception) of the STATUS register. For the BAUD RATE, the expressions to use are:

- $BAUDRATE = \frac{clock_frequency}{DIVISOR+1}$
- $DIVISOR = \frac{clock_frequency}{BAUDRATE} - 1$

where the clock frequency is 50 MHz. Given that the value of the DIVISOR to be specified in the corresponding register is an integer number, you may not achieve the exact desired BaudRate, but this is usually not a problem as long as it is within a few percent of tolerance at the low UART rates. Since the DIVISOR is on 16 bits, there is also a lower bound to the minimum Baud Rate; try to compute the value of the DIVISOR to obtain a Baud Rate of 300, and determine how many bits would be needed.

All registers are 16 bits wide, but their addresses are at multiples of 32 bit. The addresses are therefore:

Register	Address
rxdata	0x08001060
txdata	0x08001064
status	0x08001068
control	0x0800106C
divisor	0x08001070

Normal pointer arithmetic in C code using `int *` pointers should nicely handle baseaddress + offset expressions correctly, but you may want to verify their values in the initial phase of the software design to be sure. Note that the executable code that is generated from standard C uses the normal load and store assembler instructions (`ldw` and `stw` in Nios II assembler), that exploit the possible presence of a cache memory; however, register peripherals should not be cached because they may be potentially changed by the peripheral itself, leading to an inconsistency with the cache. So either don't include a cache in your processor (the one I provided does not include a cache) or you need to use special load and store instructions to bypass the cache (`ldwio` and `stwio` in Nios II assembler, the BSP that is automatically generated includes some macros that use these special instructions¹).

Hardware Design

The hardware design is identical to the one already used in the second laboratory session. All blocks (Nios II and ARM processors and their respective peripherals) are the same, but using SW9 (the left most slide switch on the board) you can select to bring the UART lines to the GPIOs (like in the second laboratory session for the software UART project) or to the Nios II UART Peripheral. If the slide switch is on position 0, close to the border

¹ Note that Nios II architecture uses Memory Mapping for the peripherals. Despite their name that may cause some confusion, the `ldwio` and `stwio` only bypass the cache, and do not drive any additional wire like what happens in the I/O Mapping technique, for instance in the x86 architecture.

of the board, you choose the GPIOs; otherwise, if the slide switch is on position 1, close to the red LED9, you choose the UART peripheral. **For this laboratory session you should set the slide switch SW9 to position 1.**

As for the second laboratory session, simply download the project, extract it to a folder (remember to avoid spaces and parenthesis in the full folder path) and open the contained project. You can look the VHDL source code, the Platform Designer project and the Chip Planner. Remember that before switching on the board you need to insert the MicroSD Card containing the preloader that routes the UART signals to the FPGA fabric rather than to the ARM processor. Then open the Programmer and configure the board.

Projects #1 to #4 test the UART transmission from the DE1-SoC board to the PC. You can show the UARTTX signal on the oscilloscope by taking it from the left header connector, top-left pin. Configure the trigger on the falling edge to capture a START BIT, and set the oscilloscope in Single Mode (you are not showing a periodic waveform, so you need to capture it and then stop). If you need, your software can drive pins of the right header connector and show them on the oscilloscope, as in the second laboratory session.

Projects #5 to #9 test the UART transmission from the PC to the DE1-SoC board. You can show the UARTRX signal on the oscilloscope by taking it from the left header connector, top-right pin. Configure the trigger on the falling edge to capture a START BIT, and set the oscilloscope in Single Mode (you are not showing a periodic waveform, so you need to capture it and then stop). If you need, your software can drive pins of the right header connector and show them on the oscilloscope, as in the second laboratory session.

A diagram of the left connector and right connectors is in Figure 2. Pay particular attention at pin numbers and GPIO indexes, because they differ for two reasons:

- The pin numbers start from 1, while the GPIO indexes start from 0.
- Power supply and ground pins are not associated with GPIO indexes.

You can optionally connect to the ground signal on the VGA connector, pin 5 (see Figure 2, on the right, pin numbers are written on the connector itself). The VGA connector is female, so it is easier to simply insert a wire rather than using a female-to-male cable as for the header connector.

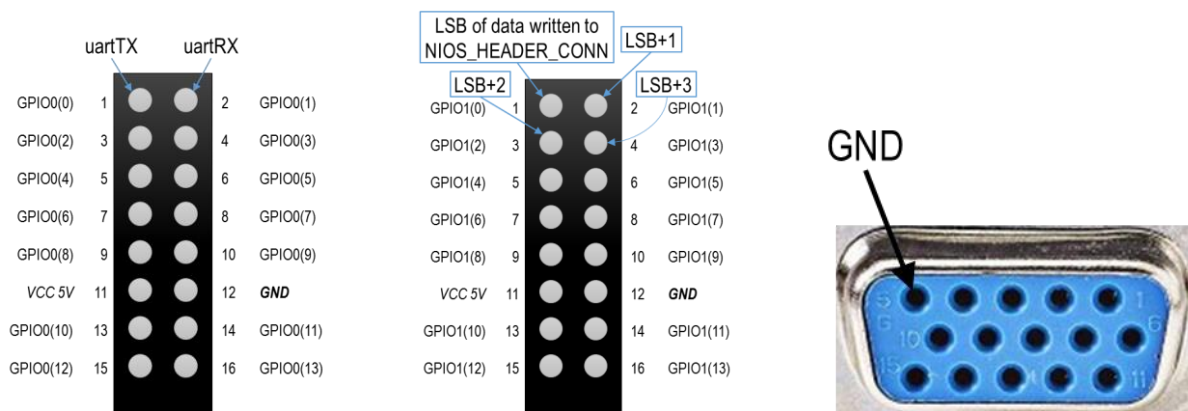


Figure 2: Left and right header connectors and the VGA connector on the DE1-SoC board

Software

The lab assignments require you to write C code for the Nios II processor and its UART peripheral. Start the Nios II Software Build Tools for Eclipse and create a new project, as in the second laboratory session. However, you must disable the Altera UART driver for the peripheral `uart_0`, or otherwise it may interfere with your code. When you have your software project open in the Nios II SBT for Eclipse, right click on the BSP project and choose Nios II → BSP Editor.... Here, under the Drivers tab, deselect (right column with the title Enable) the `altera_avalon_uart_driver` driver for `uart_0`. Then save the BSP and regenerate it. You may need to generate it twice to update all makefiles of the project.

While in the BSP Editor, also check in the Main tab that `timer_0` is assigned to the `timestamp_timer` and that the `sys_clk_timer` is set to none. In order for `printf` to work, you also must assign `stdout` to `jtag_uart_0`, and it is convenient to make the same assignment to `stdin` and `stderr`. Note that the JTAG_UART is automatically managed by the Altera library, and it is not the same UART peripheral that you need to configure and use.

When developing software, sometimes it is hard to understand the warning and error messages from the cross-compiler in Nios II SBT, because the Console is cleared, and the Problems tab does not report all information. You may then want to modify the following preferences to change this behavior: go to Window → Preferences → C/C++ → Build → Console and deactivate Always clear console before building. I also activated Wrap lines on the console. Beware that there are multiple consoles, so when you select the Console tab, you should choose the right one using a small pull-down menu on the right (it has a monitor as icon and shows the tip Display Selected Console); the one that gives the compiler messages is CDT Global Build Console.

In all software projects you need to include some header files. In particular I suggest to always include:

```
#include <stdio.h>
#include "system.h"
#include "sys/alt_timestamp.h"
#include "altera_avalon_pio_regs.h"
```

With these includes you can use `printf`, the timer and the macros to write the GPIO pins. The description of the timer functions and the macros for the GPIO is in the laboratory instructions for the second session. Just as a reminder here is a short list:

- **int** `alt_timestamp_freq(void)`
- **void** `alt_timestamp_start(void)`
- **int** `alt_timestamp(void)`
- `IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, value);`

In the proposed projects I expect you to read and write the UART registers using a pointer to the base address of the peripheral and adding the offset of the register. If you find it too difficult and see that you are losing too much time, then you can include the following file that is automatically generated in the BSP:

```
#include "altera_avalon_uart_regs.h"
```

and use these predefined macros to access the peripheral registers:

- `IORD_ALTERA_AVALON_UART_RXDATA(base)`

- IOWR_ALTERA_AVALON_UART_RXDATA(base, data)
- IORD_ALTERA_AVALON_UART_TXDATA(base)
- IOWR_ALTERA_AVALON_UART_TXDATA(base, data)
- IORD_ALTERA_AVALON_UART_STATUS(base)
- IOWR_ALTERA_AVALON_UART_STATUS(base, data)
- IORD_ALTERA_AVALON_UART_CONTROL(base)
- IOWR_ALTERA_AVALON_UART_CONTROL(base, data)
- IORD_ALTERA_AVALON_UART_DIVISOR(base)
- IOWR_ALTERA_AVALON_UART_DIVISOR(base, data)

These macros are similar to those used to access the GPIO and bypass the cache (using the `ldwio` and `stwio` assembler instructions), so they work correctly also in case a cache is present. Looking at the header file you will also see that some macros are defined to give the register offsets and correctly mask the bits of the STATUS and CONTROL registers, if you need them.

If you doubt that the compiler uses the correct pointer arithmetic, you can also look at the dump of the executable file that it generates for your code. It is in the software project folder, with extension `.objdump`. Look for the string:

<main>:

which should correspond to the beginning of your main function. All C source lines appear before the corresponding assembler code. The default compiler parameters do not apply optimizations, so it is straightforward to understand what happens. The full library also appears in this file, which is quite long.

If you want you may enable optimizations to increase your code performance or decrease its size. Select the software project in the Project Explorer on the left of the Eclipse window, and then choose Project → Properties (or right click on the project and select Properties). In the new window that appears, select Nios II Application Properties on the left, and then choose the wanted Optimization level. You can also add other parameters that you want to the gcc compiler. Note that your software project and the BSP project have distinct compiler settings (look at Nios II BSP Properties in the BSP project properties for the Optimization level parameter).

To run your program on the board, you need to compile it and download it to the SDRAM. Compilation is through the command Project → Build All. To run, use the command Run → Run Configurations... as detailed in the second laboratory session instructions.

Project #1

Write a software program on the Nios II to read registers RXDATA, TXDATA, STATUS, CONTROL and DIVISOR, and print their values on stdout.

Project #2

The UART peripheral is customized in Platform Designer to have a configurable baud rate. It means a DIVISOR register is present (register 4) to scale the clock frequency. The default value of the DIVISOR at generation time is to give a baud rate of 115200.

1. Compute the value of the DIVISOR to get a baud rate of 115200, starting from a clock frequency of 50MHz and compare it with the value obtained when reading it in Project #1.
2. Compute the value of the DIVISOR to get a baud rate of 2400. Write a software program to write the new value to the DIVISOR register, read it again, and display the new value to verify that writing was correct.

Project #3

Configure the UART peripheral for a Baud Rate of 2400. Start a terminal emulator program on a PC and configure it at 2400,8E1 (no flow control). Note that you need to enable EVEN parity. Connect the USB mini cable to the USB UART connector on the DE1-SoC board.

1. Write a software program on the Nios II to send a character over UART by writing its ASCII value in the TXDATA register and verify that it displays correctly on the terminal emulator. Print the value of the STATUS register immediately before and immediately after the transmission and explain why it is different.
If you have an oscilloscope, you can show the UARTRX line. Configure appropriately the time scale of the oscilloscope to show 3 full transmissions, so that it is easy to compare the waveforms of the next points. However, for this case you should see a single transmission only. Measure the UART clock frequency and compare it to the expected one
2. Modify the program to make two consecutive writes to the TXDATA register (with no other statement in between) and check that the two characters display correctly on the terminal emulator (send two different characters). Again, print the value of the STATUS register before the first transmission and after the second transmissions, note any differences with respect to the previous case and explain them.
If you have an oscilloscope, you should now see two consecutive transmissions, with no delay in between (the START BIT of the second transmission follows immediately the STOP BIT of the first transmission).
3. Modify the program to make three consecutive writes to the TXDATA register (with no other statement in between) and check that the result on the terminal emulator is not correct (send three different characters to distinguish which one are sent and which not). Again, print the value of the STATUS

register before the first transmission and after the last transmissions, note the differences with respect to the previous cases and explain them.

If you have an oscilloscope, you should still see two consecutive transmissions, with no delay in between (the START BIT of the second transmission follows immediately the STOP BIT of the first transmission). However, the third transmission is missing.

Project #4 (at least point 1, but do it after Project #5 if you feel you are late)

Configure the UART peripheral for a Baud Rate of 2400. Start a terminal emulator program on a PC and configure it at 2400,8E1 (no flow control). Connect the USB mini cable to the USB UART connector on the DE1-SoC board.

1. Write a software program on the Nios II to send the string "My name is" (put your name in the string) over UART (not the JTAG_UART! Don't use printf!!), and display it on the terminal emulator. For this program to work, you need to correctly handle the TRDY bit of the STATUS register (i.e. wait for it to become 1 before writing a new character to the TXDATA register).
If you want to show the entire transmission on the oscilloscope, keep in mind that it is going to take around 100ms, so set the time scale appropriately (choose 10ms/square and move the trigger instant to the left).
2. *(Optional, only if you have time, but it is funny to see the string slowly appearing on the monitor and measure the time)* Modify the string to have around 2000 characters (write some intelligent thought that comes to your mind). Estimate exactly how long it should take to transmit the entire string and then measure the real delay during the transmission. Compute the effective Bit Rate (i.e. excluding the overhead) and compare it to the Baud Rate.

How to precisely measure time:

- a. Start a chronometer and put it next to your PC display where you have the terminal emulator (it can be the chronometer of a mobile phone, showing hundredths of a second).
 - b. With a mobile phone (a second one if you used one for the chronometer) take a video of the chronometer and the terminal emulator (you may want to activate the slow motion to be more precise) and send the string of 2000 characters. Stop the video once the string is fully received.
 - c. Playback the video and take note of the value of the chronometer when the first character appears on the terminal emulator, and when the last character appears on the terminal emulator (you can pause the video or play it frame by frame to be more accurate).
 - d. Take the difference.
3. *(Optional, only if you have time)* Modify the software program in order to send the same string, but at baud rates of 9600 and 115200. Remember to modify the configuration of the terminal emulator, as well as that of your program and recompile the program. Compute again the expected latency of the transmission and measure the real delay (it is much harder now, because it is faster) in the two cases. You can try higher baud rate if supported by your hardware (e.g. 256000, 1M).

Project #5

Configure the UART peripheral for a Baud Rate of 2400. Start a terminal emulator program on a PC and configure it at 2400,8E1 (no flow control). Connect the USB mini cable to the USB UART connector on the DE1-SoC board. This project uses the polling technique to access the peripheral, so disable all interrupts by clearing

the respective bits in the CONTROL register at the beginning of the requested programs (you don't need to clear them if they are already at 0).

1. Write a software program on the Nios II to receive characters over UART by reading their ASCII value from the RXDATA register and print them to stdout. Type something on the terminal emulator and verify that the values that appear on the Nios II console are correct (use printf or the 7 segment displays). Before reading, you should constantly poll the RRDY bit in the STATUS register to make sure that there is some data available. Print the value of the STATUS register immediately before and immediately after reading the RXDATA register.
If you have an oscilloscope, show the UARTRX signal and drive a pin of the right header connector to show when the RXDATA register is actually read. You will notice that reading always happens after the full transmission (if you manage the RRDY bit correctly).
2. Repeat point 1, but remove the check of the RRDY bit of the STATUS register, and assume it is always at logic 1. See the result on the Nios II console. If you are generating a pulse on a pin of the right header connector whenever you read the RXDATA register, compare the waveforms on the oscilloscope to the previous point.
3. *(Optional, only if you have time)* Write a software program on the Nios II to receive characters over UART by reading their ASCII value from the RXDATA register and print them to stdout (insert again the check of the RRDY bit in the STATUS register), but insert a delay of a few seconds after reading one character. To implement the delay use the timer or some long processing in a loop. Type something quickly on the terminal emulator and verify that not all values appear on the Nios II Console (Receive Overrun Error). In any case, before reading, you should constantly poll the RRDY bit in the STATUS register to make sure that there is some data available. Print the value of the STATUS register immediately before and immediately after reading the RXDATA register, and explain the difference compared to the previous point. If the ROE bit is set, you need to explicitly clear it by writing 0 to the STATUS register.

Showing these transmissions on the oscilloscope is hard because a time scale of several seconds does not allow seeing the UARTRX line well. To solve the problem, many UART peripheral implement a receive and a transmit FIFO (like the UART16550, but this is not the case).

Project #6 (Optional, only if you have time)

Configure the UART peripheral for a Baud Rate of 115200. Start a terminal emulator program on a PC and configure it at 115200,8E1 (no flow control). Repeat point 1 of Project #5 with this new Baud Rate, and compare the results (and the oscilloscope waveforms if you are using it).

Project #7 (Optional, only if you have time)

Configure the UART peripheral for a Baud Rate of 2400 (or 115200). Start a terminal emulator program on a PC and configure it at 2400,8O1 (or 115200,8O1) (no flow control). Notice that now Odd Parity is requested, but the UART peripheral of the Nios II is configured for Even Parity (and it is hardwired in the peripheral; most UART peripheral would allow changing it by setting a bit in the CONTROL register). Connect the USB mini cable to the USB UART connector on the DE1-SoC board. This project uses the polling technique to access the peripheral, so disable all interrupts by clearing the respective bits in the CONTROL register at the beginning of the requested programs (you don't need to clear them if they are already at 0).

1. Write a software program on the Nios II to receive characters over UART by reading their ASCII value from the RXDATA register and print them to stdout. Type something on the terminal emulator and verify if the values are correct. Before reading, you should constantly poll the RRDY bit in the STATUS register to make sure that there is some data available. Print the value of the STATUS register after reading the RXDATA register. If the PE bit is set, you need to explicitly clear it by writing 0 to the STATUS register. Try to change the parity bit configuration of the terminal emulator without stopping your program on the Nios II and note the difference in the content of the STATUS register.

Project #8 (Optional, only if you have time)

Configure the UART peripheral and the terminal emulator program on the PC to have the same Baud Rate (it doesn't matter which one) and the same configuration. Connect the USB mini cable to the USB UART connector on the DE1-SoC board. This project uses the polling technique to access the peripheral, so disable all interrupts by clearing the respective bits in the CONTROL register at the beginning of the requested programs (you don't need to clear them if they are already at 0).

1. Write a software program on the Nios II to receive characters over UART by reading their ASCII value from the RXDATA register and store them into a C string. Declare the string to have a maximum length of 64 characters. Print the string on the Nios II console when it reaches the maximum length or when a certain character is received. You are free to decide which character to use as termination; it can be anything, like a letter, but better CR (Carriage Return), LF (Line Feed) or CTRL-D (EOF), and check the configuration of the terminal emulator to see which characters it sends when the Enter key is pressed. You can also see what happens when typing Backspace in the terminal emulator (if you really want to make it work as a backspace, you must handle it correctly in the Nios II software when filling in the string). When printing to stdout, remember to null terminate your string with character '\0', or otherwise you risk a buffer overrun.

Project #9 (Optional, only if you have time)

Configure the UART peripheral and the terminal emulator program on the PC to have the same Baud Rate (it doesn't matter which one) and the same configuration. Repeat point 1 of Project #5 but execute your code multiple times, each time increasing (or decreasing) the divisor register of the UART peripheral of the Nios II by a certain amount, while leaving the terminal emulator program on the PC with a fixed configuration. At some point the transmission should fail (you may want to check the PE Parity Error bit in the STATUS register to detect when an error occurs). Note the absolute error and the relative error of the UART generated frequency of the Nios II peripheral.

Project #10 (Optional, only if you have a lot of time, but if you are a Computer Engineering student then you may want to at least try it if you have never used interrupts)

Use interrupts on RRDY to start an Interrupt Service Routing (ISR) when receiving data. This project is more difficult compared to the previous ones, and it makes more sense when running under an operating system or at least a task scheduler, with semaphores to suspend a task and awake it when the data is available. However,

it should let you understand what is going on when using interrupts. The idea is to use an ISR to read the RXDATA register only when some data is available, as indicated by the RRDY bit of the STATUS register.

The software should consist of 2 functions:

- `main()`: it initialises variables (if any), registers the Interrupt Service Routine `uart_isr()`, configures the CONTROL register of the peripheral to generate interrupts on RRDY and the DIVISOR register for whatever Baud Rate you desire (if you leave the default, then 115200 is used). Then enters an infinite loop that increments a 32 bit variable and shows the lower 24 bits on the 7 segment displays as 6 hexadecimal digits (or the upper 24 bits if you wish, it is easier to read). Let the variable roll back to zero when it reaches the maximum value.
- `uart_isr()`: it is automatically called whenever there is an interrupt from the UART device; since only interrupts for RRDY are enabled, then this is the only possible source. So it should read the data from RXDATA register and write it to the TXDATA register (it basically echoes whatever is received), and then returns from the interrupt. You may want to check the TRDY and TMT bits of the STATUS register before transmitting data, or simply assume the transmitter is ready (you cannot wait anyway inside an interrupt service routine).

Start a terminal emulator program on a PC and configure it at the same Baud Rate of the UART peripheral. Connect the USB mini cable to the USB UART connector on the DE1-SoC board and run the software. You should see the values on the 7 segment displays increment (very quickly, the lowest digits are too fast to see them changing, but you can try taking a slow motion video), and when you type a character on the terminal emulator you should immediately see it echoed. Try implementing the same thing in polling and compare the rate of update of the counter.

To use interrupts, you need to register the Interrupt Service Routine (ISR). The Altera library already provides an Interrupt Handler that contains a table that associates an ISR function to each interrupt number defined in Platform Designer. For this project, the interrupt number given to the peripheral `uart_0` is 2 (check it in the `iRQ` column in Platform Designer, or the `#define` of `UART_0_IRQ` in `system.h` in the BSP software project). The prototype of the function to register the ISR is:

```
int alt_ic_isr_register(alt_u32 ic_id, alt_u32 irq, alt_isr_func isr, void *isr_context, void *flags);
```

where:

- `ic_id` is the ID of the Interrupt Controller (but it is ignored, pass 0);
- `irq` is the interrupt number (pass the value 2 or the macro `UART_0_IRQ`);
- `isr` is the pointer to the ISR function (if you have never used pointers to functions, this is just the name of the function that you want to register, which in my case is `uart_isr`, but you can change it if you want);
- `isr_context` is a pointer to a variable or structure that is passed to the ISR when it is called (and you need to cast it to some meaningful type to use it within the ISR; if you don't need it, pass `NULL`);
- `flags` is ignored (pass `NULL`).

The ISR that you must write has the following prototype:

```
void uart_isr(void *context);
```

As an example, see Figure 3 where you see a skeleton of the software that you should write (not really just a skeleton, it is almost complete). My code is not optimized, and I don't make checks for return values and for

status register bits (in particular RRDY and TRDY), assuming everything works correctly. If not, the code is going to have some unknown behavior

```
project9_isr.c
1 // Include files
2 #include <stdio.h>
3 #include "system.h"
4 #include "sys/alt_timestamp.h"
5 #include "altera_avalon_pio_regs.h"
6 #include "sys/alt_irq.h"
7
8 // Function prototypes
9 void uart_isr(void *context);
10
11 int main()
12 {
13     // Initialise variables
14     int *uart_ba = (int *) UART_0_BASE;
15     // Other Initialisation here
16     int statusoffs = ... ;
17     int ctrloffs = ... ;
18     int divisoroffs = ... ;
19     int br_divisor = ... ;
20
21     // Register ISR (I don't check the return value, should be 0 on success)
22     alt_ic_isr_register(0, UART_0_IRQ, uart_isr, (void *) (&uart_ba), NULL);
23
24     // Configure UART Peripheral to enable interrupts on RRDY
25     *(uart_ba + ctrloffs) = ... ;
26     // Configure Baud Rate Divisor
27     *(uart_ba + divisoroffs) = br_divisor;
28
29     // Infinite loop
30     int cnt = 0;
31     while (1) {
32         int data24bit = cnt & 0x00ffffff;
33         IOWR_ALTERA_AVALON_PIO_DATA(NIOS_7SEG_BASE, data24bit);
34         cnt++;
35     }
36
37     return 0;
38 }
39
40 void uart_isr(void *context)
41 {
42     IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, 1);
43     int *uart_ba = (int *) (*((int **) context));
44
45     // You may want to check that RRDY is at 1, if not simply return
46     // You may also want to check that TRDY is at 1, if not simply return
47     int statusoffs = ... ;
48     int status = *(uart_ba + statusoffs);
49     ...
50
51     // Read data
52     int rxdataoffs = ... ;
53     int data = *(uart_ba + rxdataoffs);
54     // Write data
55     int txdataoffs = ... ;
56     *(uart_ba + txdataoffs) = data;
57
58     IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, 0);
59     return;
60 }
```

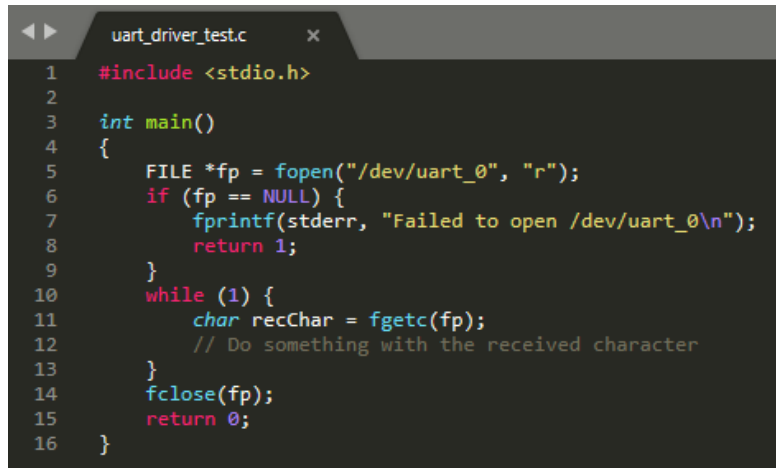
Figure 3: Example program to use interrupts to handle the UART peripheral

Project #11 (Optional, only if you have time)

Use the Altera UART Device Driver to exchange data over the UART connection. You must first enable it, as it was disabled for the previous projects. When you have your software project open in the Nios II SBT for Eclipse, right click on the BSP project and choose Nios II → BSP Editor.... Here, under the Drivers tab, enable (right

column) the altera_avalon_uart_driver driver for uart_0. Then save the BSP and regenerate it. You may need to generate it twice to update all makefiles of the project.

In your software project you can open the file /dev/uart_0 and call functions as fprintf, fscanf, fputc, fgetc, fread and fwrite. A simple example is in Figure 4. You can open the file in reading only, writing only, or in read/write mode (use the string "r+" when calling fopen in the latter case). You can repeat some of the previous projects using the device driver or design a new program, just be creative. You can use a similar approach on a linux machine with /dev/tty0 or similar if you want to experiment.



```
1  #include <stdio.h>
2
3  int main()
4  {
5      FILE *fp = fopen("/dev/uart_0", "r");
6      if (fp == NULL) {
7          fprintf(stderr, "Failed to open /dev/uart_0\n");
8          return 1;
9      }
10     while (1) {
11         char recChar = fgetc(fp);
12         // Do something with the received character
13     }
14     fclose(fp);
15     return 0;
16 }
```

Figure 4: Example program to use the Altera UART Device Driver

The driver uses interrupts to manage the UART peripheral. A full description of the driver is in my book “Analog and Digital Electronics for Embedded Systems”, although using a previous version of the interrupt API (but there are only just a few small differences). You may want to have a look at the driver source files in the BSP project; these are:

- drivers/inc/altera_avalon_uart.h
- drivers/inc/altera_avalon_uart_fd.h
- drivers/inc/altera_avalon_uart_regs.h
- drivers/src/altera_avalon_uart_init.c
- drivers/src/altera_avalon_uart_fd.c
- drivers/src/altera_avalon_uart_ioctl.c
- drivers/src/altera_avalon_uart_read.c
- drivers/src/altera_avalon_uart_write.c