Politecnico di Torino

Computer Engineering Master's Degree Course (LM-32)

Report #3

UART HW

Team members
S354939 - Pietro Levo
S358481 - Sandro Marghella
S359182 - Gianluca Rivagovernanda
S361451 - Cristina Rizzo

Professor
Claudio Passerone

November 25th, 2025

1 Introduction

In this laboratory session, the main objective was to handle the UART protocol dealing directly with the hardware registers. The Cyclone V implements the UART protocol for both receiving and transmitting data, it interfaces with a terminal emulator program on a PC, as shown in Figure 1. The same configuration of the previous laboratory (laboratory #2) was used, except for the following changes

1. The *altera_avalon_uart_driver* was disabled in the Board Support Package (BSP)
2. The physical switch #9 (SW9) was set to the 1 position

The UART was configured accordingly to the following parameters:

- Baud Rate changed during the projects (default: 115200)
- Data bits = 8
- Stop bits = 1
- Parity = Even

The registers showed in Figure 2 were manipulated to handle the UART behaviour.
As reported by the *Embedded Peripherals IP User Guide* the addresses of these registers are

- rxdata - 0x08001060
- txdata - 0x08001064
- status - 0x08001068
- control - 0x0800106C
- divisor - 0x08001070

2 Projects

These are the preprocessor macros and functions used for the implementation of all projects.

```
#define UART_0_BASE 0x8001060

#define UART_0_RX_REG (UART_0_BASE + 0)
#define UART_0_TX_REG (UART_0_BASE + 4)
#define UART_0_ST_REG (UART_0_BASE + 8)
#define UART_0_CNT_REG (UART_0_BASE + 12)
#define UART_0_DIV_REG (UART_0_BASE + 16)

void print_bin(uint8_t x) {
  for (int i = 7; i >= 0; i--) {
    printf("%d", (x >> i) & 1);
  }
  printf("\n");
}

void print_bin16(uint16_t x) {
  for (int i = 15; i >= 0; i--) {
    printf("%d", (x >> i) & 1);
  }
  printf("\n");
}
```

2.1 Project #1

```c
int main() {

  uint16_t rxdata = *((volatile uint32_t *)UART_0_RX_REG) & 0xFFFF;
  uint16_t txdata = *((volatile uint32_t *)UART_0_TX_REG) & 0xFFFF;
  uint16_t stdata = *((volatile uint32_t *)UART_0_ST_REG) & 0xFFFF;
  uint16_t cntdata = *((volatile uint32_t *)UART_0_CNT_REG) & 0xFFFF;
  uint16_t divdata = *((volatile uint32_t *)UART_0_DIV_REG) & 0xFFFF;

  printf("RX DATA: ");
  print_bin16(rxdata);

  printf("TX DATA: ");
  print_bin16(txdata);

  printf("STATUS DATA: ");
  print_bin16(stdata);

  printf("CONTROLLER DATA: ");
  print_bin16(cntdata);

  printf("DIV DATA: ");
  print_bin16(divdata);

  return 0;
}
```

The printed registers are:

Register	Value
RX DATA	0000000000000000
TX DATA	0000000000000000
STATUS DATA	0000000001100000
CONTROLLER DATA	0000000010000000
DIV DATA	0000000110110001

Table 1: UART register values observed during the experiment

These values can be compared with the configuration shown in Figure 2,

2.2 Project #2

To compute the proper DIVISOR value for a specific baud rate it is used:

$$DIVISOR = \frac{f_{clk}}{BR} - 1 \tag{1}$$

Provided $f_{clk} = 50\ MHz$ and $BR = 115200$, the DIVISOR will be $0x1B1$.
In the second scenario (BR = 2400), the divisor will be higher: $0x1428$.
In the example below is shown how to set the BR in the UART_DIVISOR register.

```c
int main() {
  volatile uint32_t *ptr_div = (volatile uint32_t *)UART_0_DIV_REG; // DIVISOR address
  uint16_t divdata = (FREQ/BAUDRATE)-1;
  *ptr_div = divdata;

  return 0;
}
```

2.3 Project #3

The goal of this project is to explore the UART transmission STATUS register behaviour in different scenarios.

2.3.1 Scenario #1

In the first scenario, the aim is to send a single character through the transmission data register, as it can be seen in line 26 of the code in Listing 1, and then compare the STATUS register before and after the transmission, as illustrated in lines 19–20 and 28–29 in the code in the Listing 1.

```
1  #define BAUDRATE 2400
2  #define PARITY EVENPPARITY
3
4  int main() {
5    char msg = 'f';
6
7    volatile uint32_t *ptr_rx = (volatile uint32_t *)UART_0_RX_REG;
8    volatile uint32_t *ptr_tx = (volatile uint32_t *)UART_0_TX_REG;
9    volatile uint32_t *ptr_sts = (volatile uint32_t *)UART_0_ST_REG;
10   volatile uint32_t *ptr_cnt = (volatile uint32_t *)UART_0_CNT_REG;
11   volatile uint32_t *ptr_div = (volatile uint32_t *)UART_0_DIV_REG;
12
13   // Configuration
14
15   // Set baudrate
16   *ptr_div |= (uint16_t)0x5160;
17
18   // 1st point
19   printf("Status before transmission: ");
20   print_bin16((uint16_t)*ptr_sts);
21
22   // wait trdy
23   // while (((*ptr_sts) & (1 << 6)) == 0);
24
25   // send msg
26   *ptr_tx = (uint32_t)msg;
27
28   printf("Status after transmission: ");
29   print_bin16((uint16_t)*ptr_sts);
30
31   return 0;
32 }
```

Listing 1: SCENARIO_3_1

The experiment's terminal output is the following:

```
Status before transmission: 0000000111101000
Status after transmission: 0000000111001000
```

Listing 2: Terminal output

The STATUS register illustrates the connection's current state through the use of flags. In the first sampling, the active flags are:

- **ROE**: stands for "Receiver Overrun Error" and is active because in our code we never read the RECEIVER data register or reset our UART through the CONTROL register, so this error and the **E** flag are not cleared;
- **TMT**: stands for "Transmitter Empty", indicating that no transmission is in progress;
- **TRDY**: stands for "Transmitter Ready" and indicates that the transmission data register is empty, ready to be overwritten by a new byte;
- **RRDY**: stands for "Receiver Ready" and indicates that the receiver data register is empty, ready to be overwritten by a new byte;
- **E**: stands for "Error" and indicates a general error.

In the second sampling, the active flags are:

- ROE;
- TRDY;
- RRDY;
- E.

It can be observed that the TRDY flag is no longer active after the character is written to the transmission data register in line 26 of the code (Listing 1). Figure 3a shows the transmission of the character via the UART. By measuring the distance between two waveforms, the actual UART clock frequency can be determined. In our case:

$$f_{clkUART} = \frac{1}{1 * \frac{2}{5} \cdot 10^{-3}} = 2500\,\text{Hz} \tag{2}$$

It is slightly different from the effective one, which is 2400 Hz.

2.3.2 Scenario #2

In the second scenario, the behaviour of the STATUS register during two consecutive transmissions is examined. To this end, the following code was written:

```
1  int main() {
2    char msg = 'f';
3
4    volatile uint32_t *ptr_rx = (volatile uint32_t *)UART_0_RX_REG;
5    volatile uint32_t *ptr_tx = (volatile uint32_t *)UART_0_TX_REG;
6    volatile uint32_t *ptr_sts = (volatile uint32_t *)UART_0_ST_REG;
7    volatile uint32_t *ptr_cnt = (volatile uint32_t *)UART_0_CNT_REG;
8    volatile uint32_t *ptr_div = (volatile uint32_t *)UART_0_DIV_REG;
9
10   // Configuration
11
12   // Set baudrate
13   *ptr_div |= (uint16_t)0x5160;
14
15   // ! USE OSCILLOSCOPE
16
17   printf("Status before transmission: ");
18   print_bin16((uint16_t)*ptr_sts);
19
20   *ptr_tx = (uint32_t)(msg + 0);
21   *ptr_tx = (uint32_t)(msg + 1);
22
23   printf("Status after transmission: ");
24   print_bin16((uint16_t)*ptr_sts);
25
26   return 0;
27 }
```

Listing 3: SCENARIO_3_2

In the code, we send two consecutive characters ("f" and "g" in lines 20–21) through the UART without waiting for the previous transmission to finish.
The output is: In the first sampling, the active flags are:

```
Status before transmission: 0000000111111100
Status after transmission: 0000000110011100
```

Listing 4: Terminal output

- **BRK**: stands for "Break Detect" and takes care of the UART's reception. It appears because we neither read the receiver data register nor reset our UART through the CONTROL register;

- ROE;
- **TOE**: stands for "Transmitter Overrun Error" and appears because in our code we never reset our UART through the CONTROL register, so we overwrite our character in the transmission register;
- TMT;
- TRDY;
- RRDY;
- E.

While in the second sampling we have only these flags active:

- BRK;
- ROE;
- TOE;
- RRDY;
- E.

The same warnings as in code 1 are present, with the addition of TOE and BRK, due to the lack of control and reset of the UART transmission in the code. As a result, the second character overwrites the first one because it is transmitted too quickly and without checking the TRDY flag. However, the oscilloscope image in Figure 3b shows that the board successfully transmits the two characters; between them, only the parity bit of "f" and the start bit of "g" are observed.

2.3.3 Scenario #3

In the last scenario, Scenario 2 is repeated, but three characters are transmitted instead of two (as shown in lines 20–21 of 5).

```
1  int main() {
2    char msg = 'f';
3
4    volatile uint32_t *ptr_rx = (volatile uint32_t *)UART_0_RX_REG;
5    volatile uint32_t *ptr_tx = (volatile uint32_t *)UART_0_TX_REG;
6    volatile uint32_t *ptr_sts = (volatile uint32_t *)UART_0_ST_REG;
7    volatile uint32_t *ptr_cnt = (volatile uint32_t *)UART_0_CNT_REG;
8    volatile uint32_t *ptr_div = (volatile uint32_t *)UART_0_DIV_REG;
9
10   // Configuration
11
12   // Set baudrate
13   *ptr_div |= (uint16_t)0x5160;
14
15   // ! USE OSCILLOSCOPE
16
17   printf("Status before transmission: ");
18   print_bin16((uint16_t)*ptr_sts);
19
20   *ptr_tx = (uint32_t)(msg + 0);
21   *ptr_tx = (uint32_t)(msg + 1);
22   *ptr_tx = (uint32_t)(msg + 2);
23
24   printf("Status after transmission: ");
25   print_bin16((uint16_t)*ptr_sts);
26
27   return 0;
28 }
```

Listing 5: SCENARIO_3_3

The terminal outputs the following data:

```
Status before transmission: 0000000111111100
Status after transmission: 0000000110011100
```

Listing 6: Terminal output

We observe the same flag activations as in Scenario 2. However, by looking at the oscilloscope trace in Figure 3c, we notice that the UART does not transmit the character "g" after "f", but instead sends "h".
This happens because the UART transmits only two characters: the first one ("f") and the last value written into the transmission register ("h"). The intermediate character ("g") is overwritten before the UART becomes ready to load a new byte. Since the code does not check the TRDY flag before writing to the transmission register, the second write occurs while the UART is still busy, causing the byte "g" to be lost.

2.4 Project #4

2.4.1 Scenario #1

This project implements multiple chars transmissions. Previously, messages were sent without waiting the TRDY flag in the control register, which made the procedure prone to communication errors.

```
int main() {
  char msg[50] = "My name is Gianluca";

  volatile uint32_t *ptr_tx = (volatile uint32_t *)UART_0_TX_REG;
  volatile uint32_t *ptr_sts = (volatile uint32_t *)UART_0_ST_REG;

  // Configuration

  *ptr_div |= (uint16_t)0x5160; // This configures BR to 2400

  printf("Status before transmission: ");
  print_bin16((uint16_t)*ptr_sts);

  for (int i = 0; i < strlen(msg); i++) {

    // Wait trdy
    while (((*ptr_sts) & (1 << 6)) == 0);

    // Send msg
    *ptr_tx = (uint32_t)(msg[i]);
  }

  printf("Status after transmission: ");
  print_bin16((uint16_t)*ptr_sts);

  return 0;
}
```

By setting the time scale to 10 ms, as shown below, the entire UART transmission is visible in Figure 4a

2.5 Project #5

This project deals again with the UART peripheral of the DE1-Soc board, this time polling technique was chosen.
So as requested the baudrate was configured to 2400 and the interrupts were disabled through CONTROL register.

To monitor the device PuTTY can be used, without flow control and obiously configured to the same baudrate of UART(PuTTY config: 2400,8E1).

2.5.1 Scenario #1

The code below can read the ASCII values present in RXDATA register and print the corresponding characters to stdout. It can be noticed that it continuosly check the RRDY bit in the STATUS register in order to see if data are available.

```
#define RRDY_MASK       (1 << 7)   // Receive Ready
#define TRDY_MASK       (1 << 6)   // Transmit Ready

int main() {
  IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, 0);
  volatile uint32_t *ptr_rx  = (volatile uint32_t *)UART_0_RX_REG;
  volatile uint32_t *ptr_tx  = (volatile uint32_t *)UART_0_TX_REG;
  volatile uint32_t *ptr_sts = (volatile uint32_t *)UART_0_ST_REG;
  volatile uint32_t *ptr_cnt = (volatile uint32_t *)UART_0_CNT_REG;
  volatile uint32_t *ptr_div = (volatile uint32_t *)UART_0_DIV_REG;

  //baudrate = 2400
  uint16_t divdata = 0x5160;
  *ptr_div = divdata;

  //disable interrupt
  *ptr_cnt = 0;

  printf("UART configured at 2400 baud, 8E1, polling.\n");
  while (1) {
      //print STATUS before reading
      printf("STATUS before RX: ");

      print_bin16((uint16_t)(*ptr_sts));
      //Polling: wait for RRDY to be set

      while (((*ptr_sts) & RRDY_MASK) == 0) {
          // busy wait
      }
      IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, 1);
      //read RXDATA
      uint8_t rxchar = (uint8_t)(*ptr_rx & 0xFF);
      IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, 0);

      //print STATUS after the reading
      printf("STATUS after RX:  ");
      print_bin16((uint16_t)(*ptr_sts));

      //print the received character
      printf("Received Character: '%c' (ASCII=%d)\n", rxchar, rxchar);
  }
  return 0;
}
```

As shown above, the program prints the status register before and after the reading.

```
STATUS before RX: 0000000111111100
STATUS after RX:  0000000101111100
Received Character: 'g' (ASCII=103)
```

On the oscilloscope are plotted first the UARTRX and second the RRDY pulse, the latter is received at the end of the full transmission (Figure 5a).

2.5.2 Scenario #2

Now the above exercise is repeated but this time assuming the RRDY bit of the status register is always set and so it is not necessary to check it.

```c
int main() {
  IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, 0);
  volatile uint32_t *ptr_rx  = (volatile uint32_t *)UART_0_RX_REG;
  volatile uint32_t *ptr_sts = (volatile uint32_t *)UART_0_ST_REG;
  volatile uint32_t *ptr_cnt = (volatile uint32_t *)UART_0_CNT_REG;
  volatile uint32_t *ptr_div = (volatile uint32_t *)UART_0_DIV_REG;

  //baudrate 2400
  uint16_t divdata = 0x5160;
  *ptr_div = divdata;

  //Disable interrupt
  *ptr_cnt = 0;

  printf("UART configured at 2400 baud, 8E1, NO polling RRDY.\n");
  int freq = alt_timestamp_freq();
  const int DELAY_TICKS = freq/2;

  while (1) {
    IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, 1);

    /*alt_timestamp_start();
    while (alt_timestamp() < DELAY_TICKS);
    */

    //read RXDATA without checking RRDY
    uint8_t rxchar = (uint8_t)(*ptr_rx & 0xFF);
    IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, 0);

    //print received character
    printf("Received Character: '%c' (ASCII=%d)\n", rxchar, rxchar);
  }
  return 0;
}
```

Looking to the oscilloscope, a pulse is received after every full transmission (Figure 5b).

A Appendix

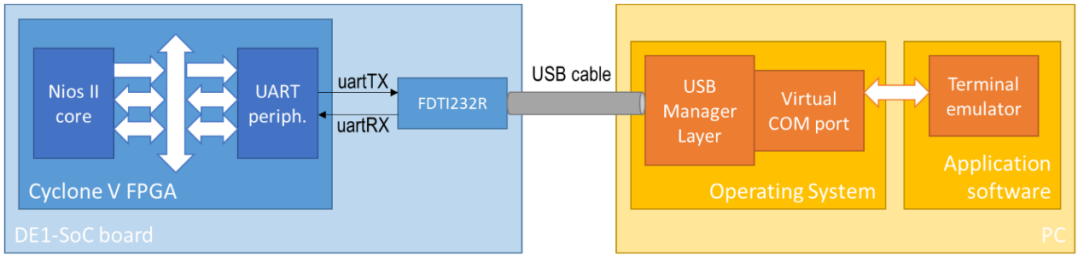
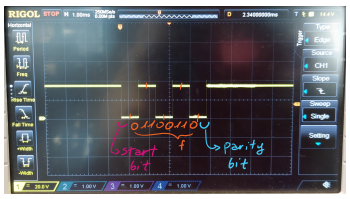


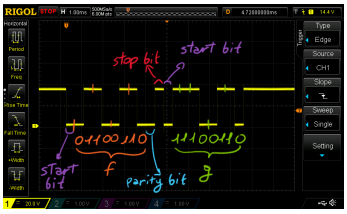
Figure 1: UART HW-SW

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													

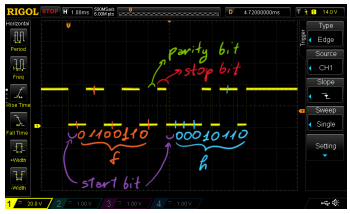
Figure 2: UART register configuration.



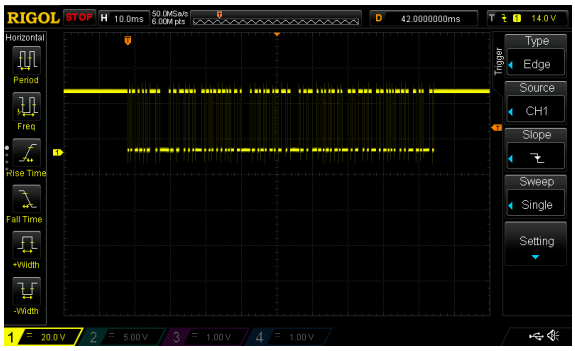
(a) Project 3 Scenario 1



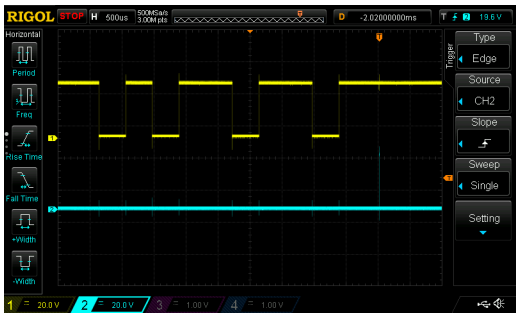
(b) Project 3 Scenario 2



(c) Project 3 Scenario 3



(a) Project 4



(a) Project 5 Scenario 1

(b) Project 5 Scenario 2