Electronics for Embedded Systems

10/11/2025, 11/11/2025 Laboratory

In this laboratory you will learn how to configure and program a Nios II processor. The goal is to connect the Nios II processor running on the Altera DE1-SoC board with a desktop PC, using a serial interface based on the UART protocol for communication.

Hardware Design

The hardware system is available as a Quartus Prime project. Download the uarthwsw.zip file and extract it in a directory where you have write permissions. The entire path of the final directory should not contain any space, or eventually the tools will fail with some obscure error message. Start Quartus Prime and select File → Open Project… to open the project …/uarthwsw/uarthwsw.qpf.
The hardware design is already compiled with all correct pin assignments. In principle you can program the board and proceed to the software design directly; however, it is good to take a look at the project, to avoid issues while developing the software afterwards. Spend 10 minutes to examine the hardware project, and then proceed to the software (in this way you know where to look in case you need it). In particular:

- Open the VHDL source file uarthwsw.vhd that you find in the project directory. It contains the top level entity called uarthwsw, with many defined input and output ports. Some of them are commented out (this entity basically contains all the possible ports of the Cyclone V FPGA on the DE1-SoC, with names that reflect their functions; the file pin_assignment_DE1_SoC.tcl, which is also available in the project directory, performs all the correct pin assignments for these port names). The relevant pins are:
  a. CLOCK_50: it is the input clock, connected to the 50 MHz quartz on the board;
  b. DRAM_*: these are connected to the 64 Mbytes SDRAM on the board;
  c. HEX*_N: connected to the 7 segment displays on the board;
  d. KEY_N: connected to the 4 active low pushbuttons on the board;
  e. LEDR: connected to the 10 LEDs on the board;
  f. SW: connected to the 10 slide switches on the board;
  g. VGA_*: connected to the DAC to drive the VGA connector;
  h. GPIO_0: connected to the left header connector on the board (don't confuse it with a GPIO peripheral inside a processor, although a GPIO peripheral is normally used to drive these signals);
  i. GPIO_1: connected to the right header connector on the board;
  j. HPS_UART_RX: connected to the UART receive signal on the board;
  k. HPS_UART_TX: connected to the UART transmit signal on the board;
  l. HPS_*: other HPS (ARM9 processor) signals, not used in this laboratory session.

- Still in the VHDL source file uarthwsw.vhd, look at the architecture for entity uarthwsw that includes three components and some signal assignments:
  a. The Platform Designer generated system nios_hps_system, that includes the Nios II processor and the HPS (ARM9 processor); it is instantiated as u0, with all proper port maps.
  b. A decoder for the 7 segment displays (from 4-bit hex value to 7 signals). It is instantiated 6 times, one for each 7 segment display present in the DE1-SoC board.
  c. The VGA oscilloscope component; it is instantiated as v0, with all proper port maps.
  d. Some signal assignments to define how to connect UART lines to the Nios II processor (bypassing the ARM9 processor; these signals are called LOANIO) and to route some signals to the header connectors to display them on the oscilloscope.

- Open Platform Designer with Tools → Platform Designer and load the nios_hps_system.qsys file. Here you find all components that are part of the u0 block instantiated in the architecture, with the interconnections, using a textual representation. There are more components than what is actually needed for this laboratory session; the relevant ones are:
  a. The PLL pll_0 to generate proper clocks for the processor and the external SDRAM.
  b. The Nios II processor nios2_qsys_0 (classic processor, economy /e version, no cache, no pipeline, very slow).
  c. The SDRAM Controller sdram_controller_0 (it is a peripheral for the Nios II processor).
  d. The GPIO nios_uartrx to read the status of the input UART line.
  e. The GPIO nios_uarttx to set the status of the output UART line (use it only in the optional project #9 to send a reply from the Nios II processor to the terminal emulator).
  f. The GPIO nios_header_conn to drive the right header connector on the DE1-SoC board (the left header connector is directly driven from the VHDL signal assignments).
  g. The timer timer_0 to measure time in the software.
  h. The JTAG UART jtag_uart_0 to provide stdin, stdout and stderr to the software program (so that printf actually works).

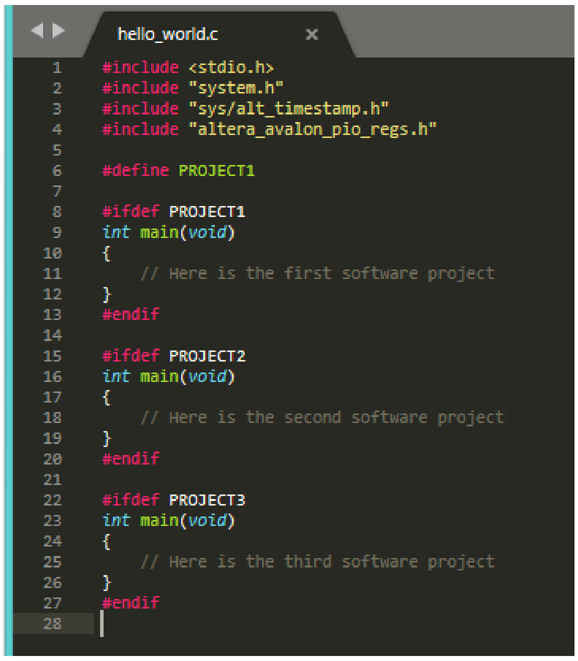
- Still in Platform Designer you can open the schematic diagram of the system with View → Schematic. This is a graphical view of the same system and interconnections. You can open the HPS and see its internal components (the two cores and all peripherals).

- Open the Chip Planner with Tools → Chip Planner and look at the synthesized system. It is much larger than the small designs developed in the first laboratory session, but most of the FPGA is still unused. Double clicking on the logic elements shows you the implementation on the LUTs, but it is impossible to separate and understand the various parts.

Before starting the software development, insert the MicroSD Card containing the preloader in the card reader (on the right hand side of the board), connect the power supply and the USB Blaster cable, and switch the board on. The ARM9 processor boots from the MicroSD Card, although you don't see anything happening. After booting the UART lines are routed to the FPGA, that still needs to be configured.
Start the Programmer with Tools → Programmer and program the board with the system (the uarthwsw.sof file is available in the output_files folder, but it might be already selected correctly). Now the software can be developed and run on the Nios II processor on the board.
Set the switch SW(9), the leftmost switch of the board, to the 0 position. This hardware design will be used in the 3rd laboratory session as well, by setting this switch to the 1 position. Keep it at 0 for the entire second laboratory session.

Software design

You will write several different software projects that will differ considerably in complexity. However, to avoid restarting from the beginning each time you write a new software program, I suggest to keep them in the same source file, and distinguish them using defines. See as an example Figure 1: by changing the #define at the beginning of the source file, you can easily select one of the software projects excluding the others. Remember to always save and recompile your software after making a change in the #define.
In all software projects, except the first one, you will have to interact with external pins of the Cyclone V FPGA and the DE1-SoC board through GPIOs, i.e. physical pins whose logic state can be read or set using a C program running on the Nios II processor implemented on the FPGA. Refer to the slides presented in class for detailed examples on how to do this. In your software you must also include some library files for everything to work, like those shown in Figure 1.



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

#define PROJECT1

#ifdef PROJECT1
int main(void)
{
    // Here is the first software project
}
#endif

#ifdef PROJECT2
int main(void)
{
    // Here is the second software project
}
#endif

#ifdef PROJECT3
int main(void)
{
    // Here is the third software project
}
#endif

```

Figure 1: Example software with multiple projects

Project #1

Write a simple program to test the processor; it is just a single printf, to check that everything is alive. The following steps are used to create a new software projects, and you don't need to repeat them for the following projects if you are simply going to append code to the same file, as illustrated in Figure 1.

1. Start the Nios II Software Build Tools for Eclipse (Quartus Prime 18.1 or above). In the desktop PC available in the laboratory, look for it in the Intel group of programs.

2. Create and choose a workspace directory. The workspace is where Eclipse stores its own information, such as the size of the windows, their placement and so on. It can be a subdirectory of the project directory that you created at the beginning of the hardware design, but it should not be called software (suggestion: call it workspace). There should be no spaces in the entire path name of the directory. Click OK. The development environment starts. It is the normal Eclipse environment, customized for the Altera Nios II processor.

3. Create a new project with File → New → Nios II Application and BSP from Template. This software project is different from the hardware project created within Quartus Prime to generate the instance of the processor, but the two projects are obviously related. In fact, the available features of the processor core need to be known to the software compiler in order to correctly generate code (e.g. presence of custom instructions, HW or SW implementation of multiplication and division). Moreover, the system is able to generate the Hardware Abstraction Layer (the BSP, Board Support Package) for a particular generated processor: that depends on the kind and number of peripherals that were instantiated. The link between the HW and the SW projects is through the SOPC description. In the new project window, specify the following:
    a. SOPC Information File name: enter the name of the file generated by Platform Designer; you can find it in the project directory, with extension .sopcinfo. If everything is fine, it should be nios_hps_system.sopcinfo.
    b. CPU name: select the name of processor core. In this case, you have only one core, so you can accept the default value (cpu_0). This option is useful only for multi-core projects.
    c. Project name: enter a name for the project.
    d. Project location: choose a location for the project, or leave the default location. This should be a directory with no spaces in its full name. The directory should not be inside the workspace directory, or you will get an error message later.
    e. Choose the project template Hello World.

Now click on Next to see the BSP options for your project, and leave the default values. Click on Finish to create the project. Two projects are actually created: the one with the name you selected, and another with the same name and the suffix _bsp appended.

4. Open the file hello_world.c of your software project, and change the default string in the printf to any that you want. Save the file, or otherwise compilation will take the previous version of it.

5. Right click on the name of the BSP project in Project Explorer, select Nios II → BSP Editor… towards the end of the context menu and then wait for the BSP Editor window to appear. Check that stdout is set to jtag_uart_0; if it is set to none, set it to jtag_uart_0 (you may want to have stdin and stderr set to jtag_uart_0, as well); then set the sys_clk_timer to none and the timestamp_timer to the name of your timer (it should be timer_0). If you made any change, save the configuration and click on Generate in the BSP Editor, then close the BSP Editor window.

6. Build the project using Project → Build All. The compilation terminates and you get a message on the Console tab at the bottom of the IDE window. Any warning or error is in the Problems tab, so check it before continuing. Try to change the source so that you don't get any warning (and obviously no error, as well). Remember that each time you make a change to the source file you need to save it, or otherwise compilation will ignore the changes. If you want to recompile everything from scratch, including the BSP, you may want to clean the project first, with Project → Clean….

7. To run the project on the DE1-SoC board, you must define a Run Configuration. Select Run → Run Configurations... and double click on Nios II Hardware. A new configuration is automatically created, and you can change its name if you will (the default is New_Configuration). You must select the Project name, but leave all the other options to the default values. Check under the Target Connection tab that the DE-SoC cable was recognized, otherwise click on the Refresh Connections button (it is on the right hand side, you may have to scroll the window to find it). If the cable is not recognized, check the connections and verify that the DE1-SoC board is on and programmed (see the last paragraph of the Hardware project). When you click on Run, the compiled software is downloaded on the board and immediately executed. After a small delay, you should see the console output on the screen.

8. Browse the content of the project and the BSP. There are many interesting files. In particular, have a look at the system.h file and at the drivers and HAL folders in the BSP, which contain all the library files for the complete application. For instance, the assembler initialization code is in HAL → src → crt0.S.

You can also run the program using a debugger. Instead of running software with Run Configurations, use the command Debug Configurations. Usual debugger functions are available, such as breakpoints in the source code, step by step execution, disassembly, etc.

Project #2 (Optional, if you want to use the oscilloscope)
Write a program to periodically complement the logic value on a pin on the right header connector, and display the result on the oscilloscope. Connect a wire from a pin of the header connector to the probe of the oscilloscope, and remember to connect also the ground terminal (to the header connector, or to pin 5 of the VGA connector of the DE1-SoC board, which is also a ground, and it is a female connector so it is easier to insert a simple wire). Between the instructions that complement the pin, insert a delay that might be due to some dummy statements (perform some useless additions, or a printf, or whatever), or use a timer to wait for a certain time.
To write data to an output PIO, use the macro:

```
IOWR_ALTERA_AVALON_PIO_DATA(NIOS_HEADER_CONN_BASE, value);
```

where value is a 32 bit integer number (only 32 pins out of 36 of the header connector can be controlled in this way, see the interconnections in the VHDL file and the pin assignments). The LSB of value changes pin 0 of the header connector, LSB+1 changes pin 1 and so on.
To use the timer, you have the following functions available:

- int alt_timestamp_freq(void): returns the number of clock ticks per second of the timer.
- void alt_timestamp_start(void): reset the timer to the initial value 0.
- int alt_timestamp(void): returns the number of clock ticks since the last call to alt_timestamp_start().

So, for instance, if you want to measure the number of ticks of a block of code, you can use the following:

```
alt_timestamp_start();
/* Some code here */
int t1 = alt_timestamp();
/* The code you want to measure here */
int t2 = alt_timestamp();
printf("Number of ticks: %d\n", t2 – t1);
```

To wait for a specified amount of time expressed as a number of clock ticks:

```
alt_timestamp_start();
while (alt_timestamp() < NUMBER_OF_TICKS) /* Do nothing */ ;
```

Using the macros for reading and writing pins and the timer functions requires to load some header files in your C code. See Figure 1 that shows the needed #include statements.

Project #3 (Optional, if you want to use the oscilloscope)
Using a USB cable with a mini type B connector, connect a PC to the UART connector of the DE1-SoC board. On the PC start a serial terminal emulator program (for instance, PuTTY, Teraterm, Realterm, minicom, …) with the following parameters:

- Baud Rate: 300
- Number of bits in a character: 8
- Parity: None
- Number of STOP BITs: 1
- Flow Control: None

Choose the COM port appropriately, maybe by checking the Device Manager (on my laptop it is usually COM3 or COM4). If you want, enable local echo on the terminal emulator. Note that the PC running the terminal emulator can be different from the PC running Quartus Prime and the Nios II Software Development Environment.
Show pin 2 that corresponds to GPIO0(1) of the left header connector on an oscilloscope; connect a wire from the pin of the header connector to the probe of the oscilloscope, and remember to connect also the ground terminal (to the header connector, or to pin 5 of the VGA connector of the DE1-SoC board, which is also a ground, and it is a female connector so it is easier to insert a simple wire). Set the trigger of the oscilloscope to fire on the falling edge. Set the horizontal scale to show approximately one bit time (around 3.3 ms) per square. Set the oscilloscope in Single mode. The oscilloscope should not show any waveform at this point.
Type a single character on the terminal emulator. A small blue LED close to the USB UART connector on the DE1-SoC should briefly light up, and the oscilloscope should show a waveform and then STOP rendering new waveforms. Compare the waveform with the expected one (use an ASCII table to determine the code that is sent over the UART line).
Try with different keys. Every time you need to rearm the Single mode on the oscilloscope. Try changing the Baud Rate or any of the other parameters, to see the differences in the displayed waveform.

Project #4
Connect a PC to the DE1-SoC board as explained in Project #3, and start a serial terminal emulation program with following parameters:

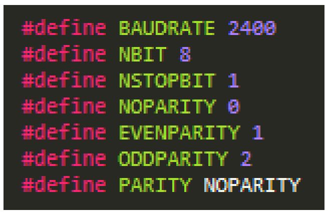
- Baud Rate: 300

- Number of bits in a character: 8
- Parity: None
- Number of STOP BITs: 1
- Flow Control: None

Write a program on the Nios II to read the UART line and decode the transmitted character. When developing your program keep in mind that you don't have floating point arithmetic, so never use float or double data types, but rather use integer arithmetic. If you have to compute a division, make sure that you don't underflow, i.e. the denominator is bigger than the numerator, as the result will be 0. Also having numerator and denominator of the same order of magnitude usually leads to big errors.
You have to read the logic state of the PIO pin connected to the UART line. You can use the following macro:

```
int val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE);
```

Remember to mask the highest bits and keep only the LSB, which is what matters for this application (the other bits are not connected and maybe undefined, although they are typically at zero). You will also need to measure time to sample the pin at the correct rate and instants; use the timestamp timer, as described in the optional Project #2 to implement the proper delays.
Try to write the software so that it is easy to change the UART parameters. For instance, use defines for the Baud Rate, the number of bits per character, the number of STOP BITs and the kind of parity. Then write the software to use those defines and compute the timing information appropriately. This makes it much easier to implement the other projects. See Figure 2 for an example.



```
#define BAUDRATE 2400
#define NBIT 8
#define NSTOPBIT 1
#define NOPARITY 0
#define EVENPARITY 1
#define ODDPARITY 2
#define PARITY NOPARITY
```

Figure 2: definitions for UART parameters

Remember that printf is extremely slow, so avoid calling it while decoding the bits of the transmission, because it significantly changes the timing. You can use it while debugging, but otherwise it should appear only before or after a complete transfer.
If you are using the oscilloscope, then you can show the UART line as in project #3, and you can toggle a pin on the right header connector when sampling is performed as in project #2, to be shown on a second channel on the oscilloscope. This greatly helps understanding failures of the code.
A couple of suggestions to increase the performance and achieve a better Baud Rate:

- Compute at the beginning all time instants after the START Bit when you will need to sample the data, and never restart the timer (restart it only after detecting the START Bit). Restarting the timer takes some time, so if you do it inside the loop to read bits, it will decrease the performance.
- Unroll the loops, so you save incrementing a variable and testing and jumping. To support a variable number of bits in the character use #ifdef instead of if construct. When you recompile, it will include only the required code and avoid testing and jumping.

You can use the flow chart at the end of the laboratory instructions as a reference for your design, but you are free to adopt any other algorithm you feel appropriate to reach your goal.

Project #5
Increase the baud rate until you find one that cause the transmission to fail. The standard baud rate values are the following: 110, 150, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600 and 115200. Other values are also possible (for instance, 28800 and 33600, or values higher than 115200 like 128000 and 256000). If you are using the oscilloscope, show where the sampling happens outside the correct sampling window.

Project #6 (Optional, only if you have time)
Try to set incompatible baud rate parameters or number of bits per character between transmitter (the PC) and receiver (your software running on the Nios II on the DE1-SoC board), and see how the transmission fails. Try

the following:

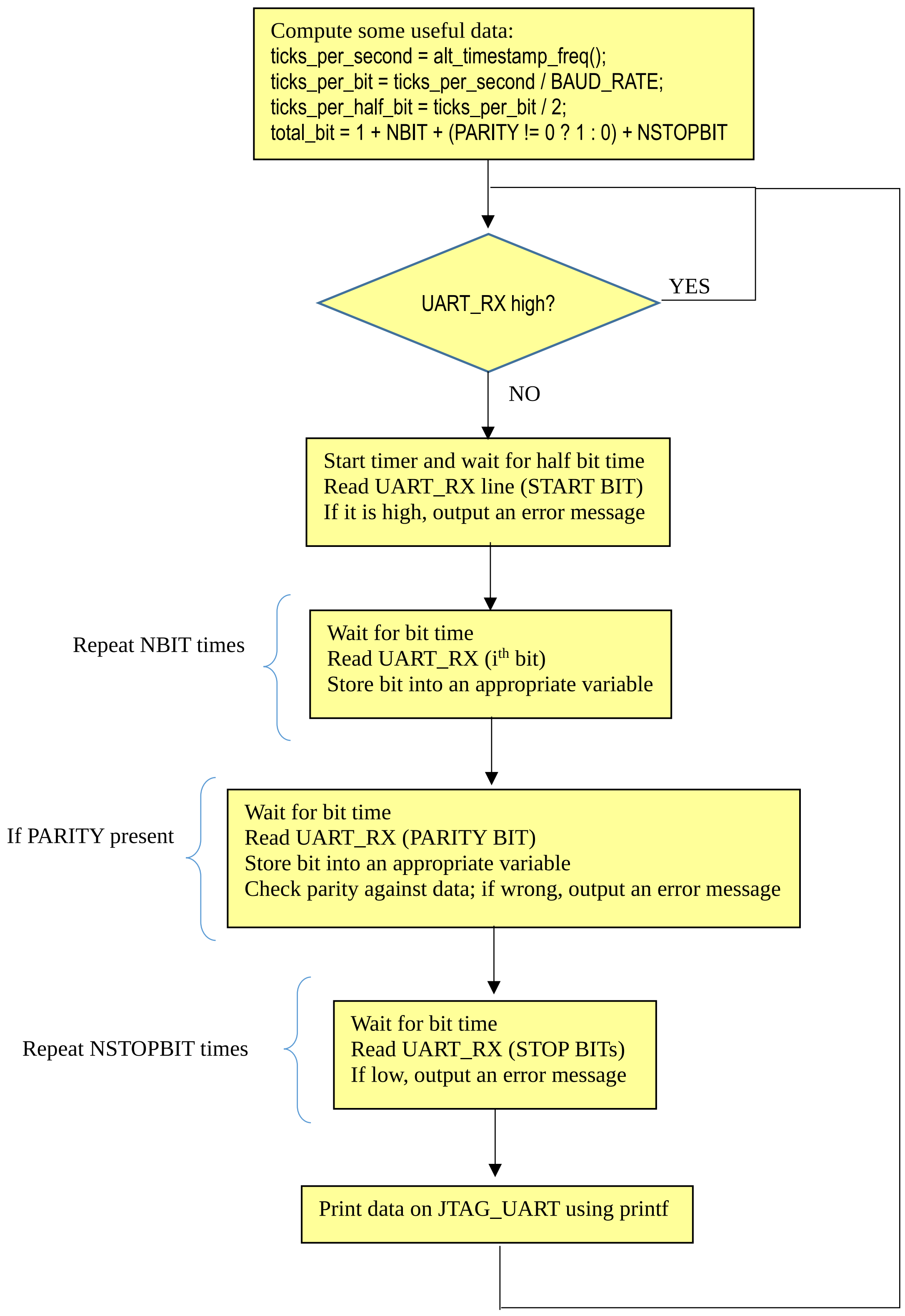
- BaurRateTX > BaudRateRX
- BaurRateTX < BaudRateRX
- nbitpercharTX > nbitpercharRX
- nbitpercharTX < nbitpercharRX

If you set incompatible number of STOP BITs, this should not determine any failure, unless a second character is sent immediately after another character, with no idle time in between.

Project #7 (Optional, only if you have time)
Choose a Baud Rate that works for your system, and change the Baud Rate of your receiving program in steps of 1% (or more), until you find a value that causes a failure. For instance, if your system works at 2400, keep the transmitter at 2400 and change the receiver to 2424, then 2448, then 2472, then 2496, and so on until it fails. Determine the relative error that causes the transmission to fail.

Project #8 (Optional, only if you have time)
Implement parity checking (odd parity, even parity). To cause a parity failure, set opposite parities in the transmitter and in the receiver (or no parity against some kind of parity).

Project #9 (Optional, only if you have time)
Implement UART transmission as a reply to the received data. For instance, you can send back the same character, but with the ASCII code incremented by one. Use the same parameters (baud rate, number of bits per character, …) as the received data. The transmitted character should appear on the serial terminal emulator, and can be shown on the oscilloscope by taking it from pin 0 of the left header connector.


Compute some useful data:
ticks_per_second = alt_timestamp_freq();
ticks_per_bit = ticks_per_second / BAUD_RATE;
ticks_per_half_bit = ticks_per_bit / 2;
total_bit = 1 + NBIT + (PARITY != 0 ? 1 : 0) + NSTOPBIT
UART_RX high?
YES
NO
Start timer and wait for half bit time
Read UART_RX line (START BIT)
If it is high, output an error message
Repeat NBIT times
Wait for bit time
Read UART_RX (ith bit)
Store bit into an appropriate variable
If PARITY present
Wait for bit time
Read UART_RX (PARITY BIT)
Store bit into an appropriate variable
Check parity against data; if wrong, output an error message
Repeat NSTOPBIT times
Wait for bit time
Read UART_RX (STOP BITs)
If low, output an error message
Print data on JTAG_UART using printf