Serial Interface

2024-09-19 · 7 min · Justin Rajewski | Suggest Changes

▶ Table of Contents

In this tutorial we will use the USB port and create a project that will echo back all the data sent to the Alchitry board. This will teach you how to use the *Component Library* in your projects.

Getting Started

Like before, we will start by creating a new project. I'm calling mine *Serial Port Echo*. It is from the *Base Project* template.

Great! Now we have a bare-bones project. The USB port is accessible to the FPGA through the FTDI USB<->serial bridge. Serial is often called *UART* (**U**niversal **A**synchronous **R**eceiver **T**ransmitter) and this is the name of the components we will need.

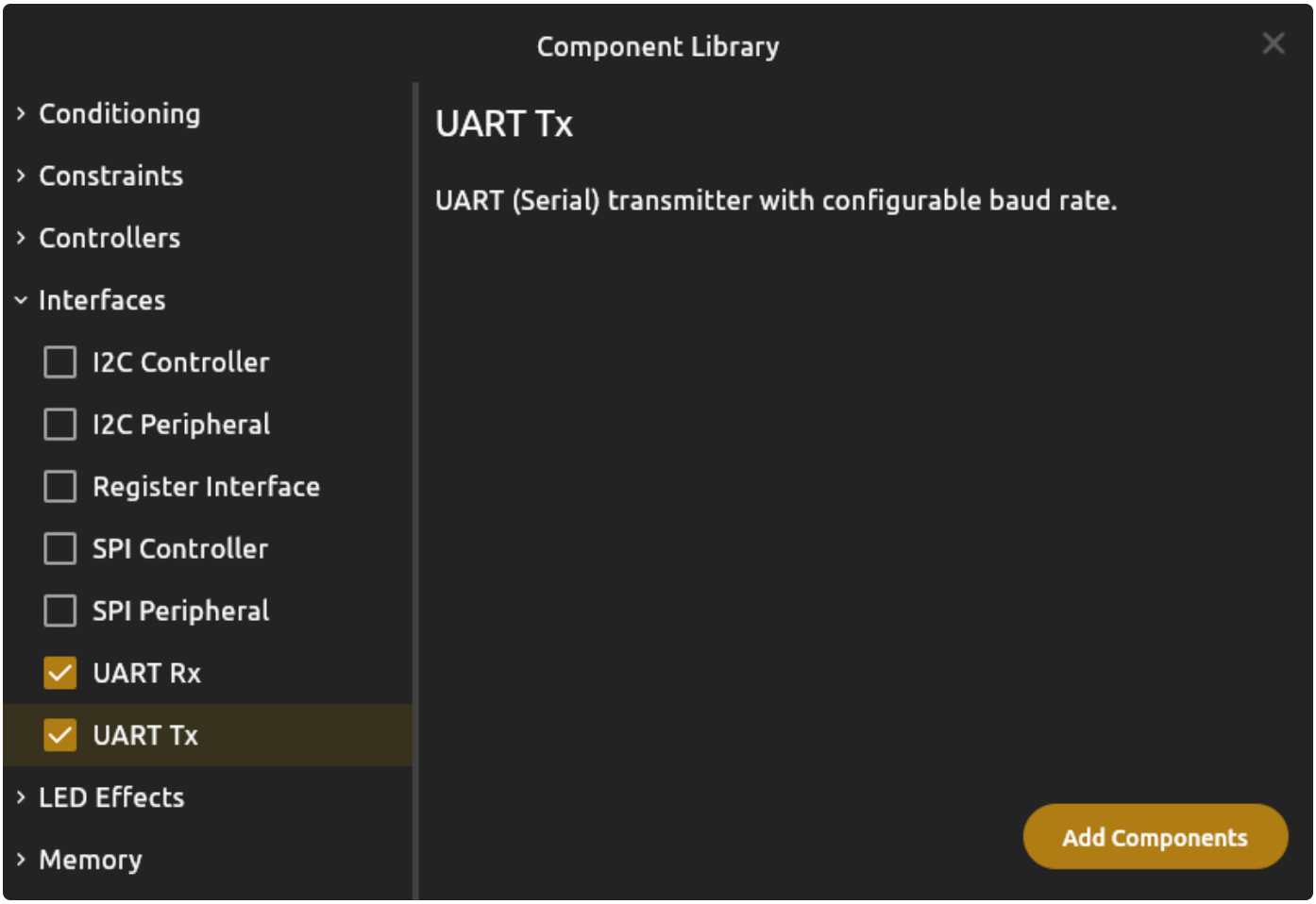
Components

Components are pre-written modules that you will likely need to use in many of your projects. For this tutorial, we need to add two components to our project. One to

receive data and one to send data.

Launch the *Component Library* by clicking the three blocks icon in the toolbar. If you have a Cu, this will open the library directly. If you have an Au, this will open a dropdown. Click *Component Library* to open the library.

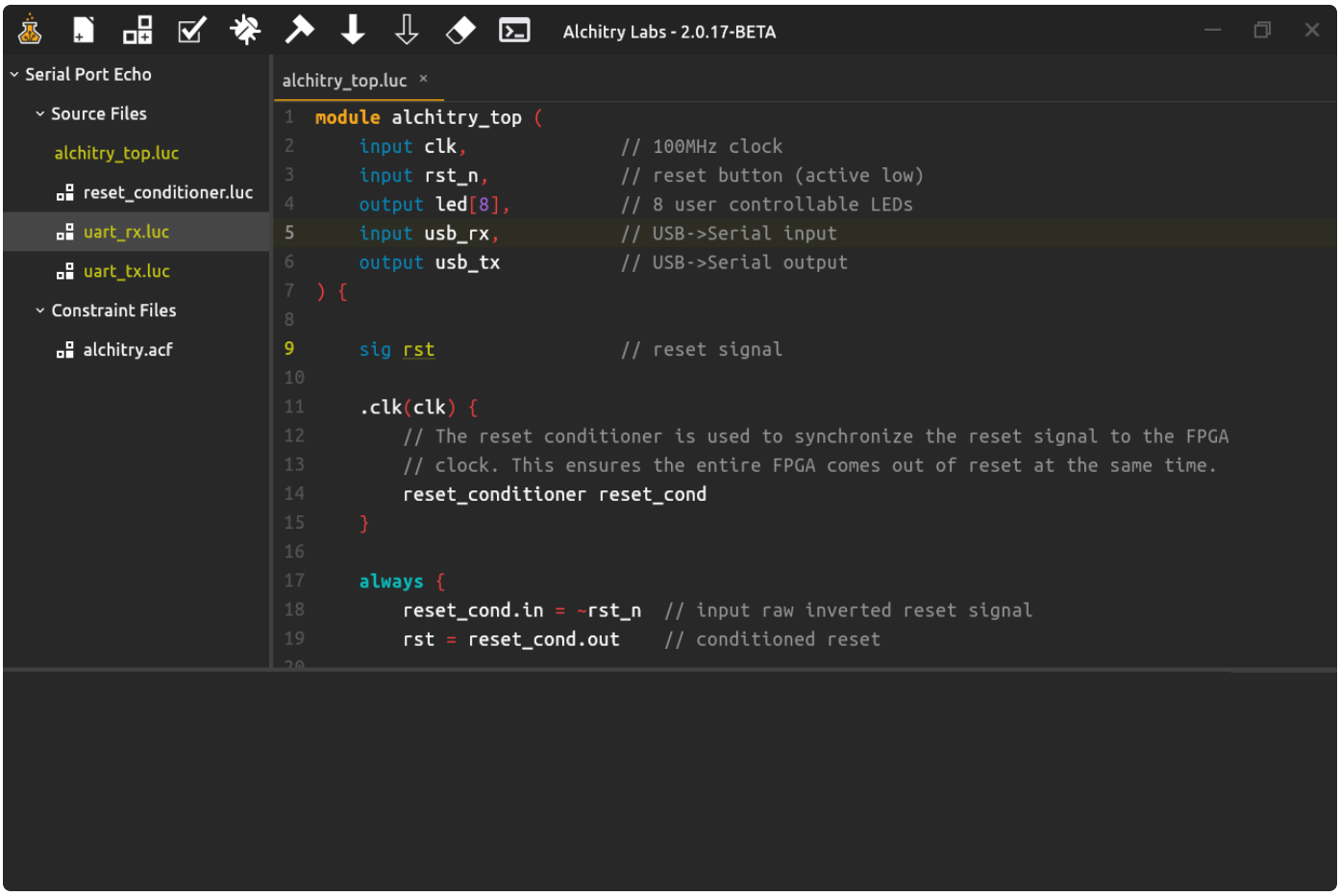
Under *Interfaces*, you will find the components *UART Tx* and *UART Rx*. Click the checkboxes next to each one.



You can click each component for a short description of what it does.

Feel free to explore the other categories to see what's available. The list of components grows over time with new releases of Alchitry Labs.

Click *Add* to add the components to your project.



Components are indicated by the three box icon next to their name.

You can open any of the components you want to take a look at how they work. However, for this tutorial we will be using them as a black box.

Instantiating the Component

Now that the components we need are in our project, we need to use them.

We are going make some changes in the top file.

```
11  .clk(clk) {
12      // The reset conditioner is used to synchronize the reset signal
13      // clock. This ensures the entire FPGA comes out of reset at the
14      reset_conditioner reset_cond
15
16      .rst(rst) {
```

```
17          uart_rx rx(#BAUD(1_000_000), #CLK_FREQ(100_000_000))
18          uart_tx tx(#BAUD(1_000_000), #CLK_FREQ(100_000_000))
19      }
```

This will create instances of `uart_rx` and `uart_tx` named `rx` and `tx` respectively.

We need to specify two parameters for each one. The `BAUD` parameter is the number of bits per second it should send. The important thing is that you match this rate to the one you set on your computer. The serial port monitor in Alchitry Labs uses 1M baud by default so that is what we specify here. The USB<->serial interface supports up to 12M baud.

Underscores can be inserted anywhere in numbers for readability.

The other parameter, `CLK_FREQ`, is the frequency of the clock. This is used to calculate how many clock cycles are required per bit.

If you only do this, you will actually get some errors. These errors are because some of the inputs to the modules were never assigned.

We are going to hook the modules up to each other so that when data is received, it is promptly sent back.

Change the always block to look the following.

```
22  always {
23      reset_cond.in = ~rst_n  // input raw inverted reset signal
24      rst = reset_cond.out    // conditioned reset
25
26      led = 8h00              // turn LEDs off
27
28      rx.rx = usb_rx          // connect rx input
29      usb_tx = tx.tx          // connect tx output
30
```

```
31      tx.new_data = rx.new_data
32      tx.data = rx.data
33      tx.block = 0            // no flow control, do not block
34  }
```

On lines 28 and 29, we connect the external input and output to our two modules.

On lines 31 and 32, we connect the `rx` module's outputs to the `tx` module's inputs.

On line 33, we set the `block` input of `tx` to `0`. When this value is `1`, the `uart_tx` component won't send any data out. This is useful if you have some way to tell that the receiver is busy. With the USB<->serial converter, we just assume the data is being read from the computer in a timely manner to keep the FTDI's buffer from overflowing. This is a reasonable assumption as long as there is some program actually reading the data.

Sending and Receiving Data

When new data arrives, the signal `rx.new_data` goes high. This tells you that the data on `rx.data` is valid. Normally you would want to wait for `rx.new_data` to go high and then do something with the data.

Writing data to the serial port follows the same idea. We set `tx.data` to the byte to send, and we set `tx.new_data` high. However, there is one more signal to look out for. That is `tx.busy`. If this signal is high, the transmitter is busy for some reason, either it is currently sending a byte or block is high. Either way, if you try to send data when this is high, it will be ignored.

For this simple example, we are going to ignore `tx.busy`. This should not be a problem since we never block and the bytes coming in arrive at the same rate we can send them out.

The next tutorial will handle this more gracefully by actually respecting this flag.

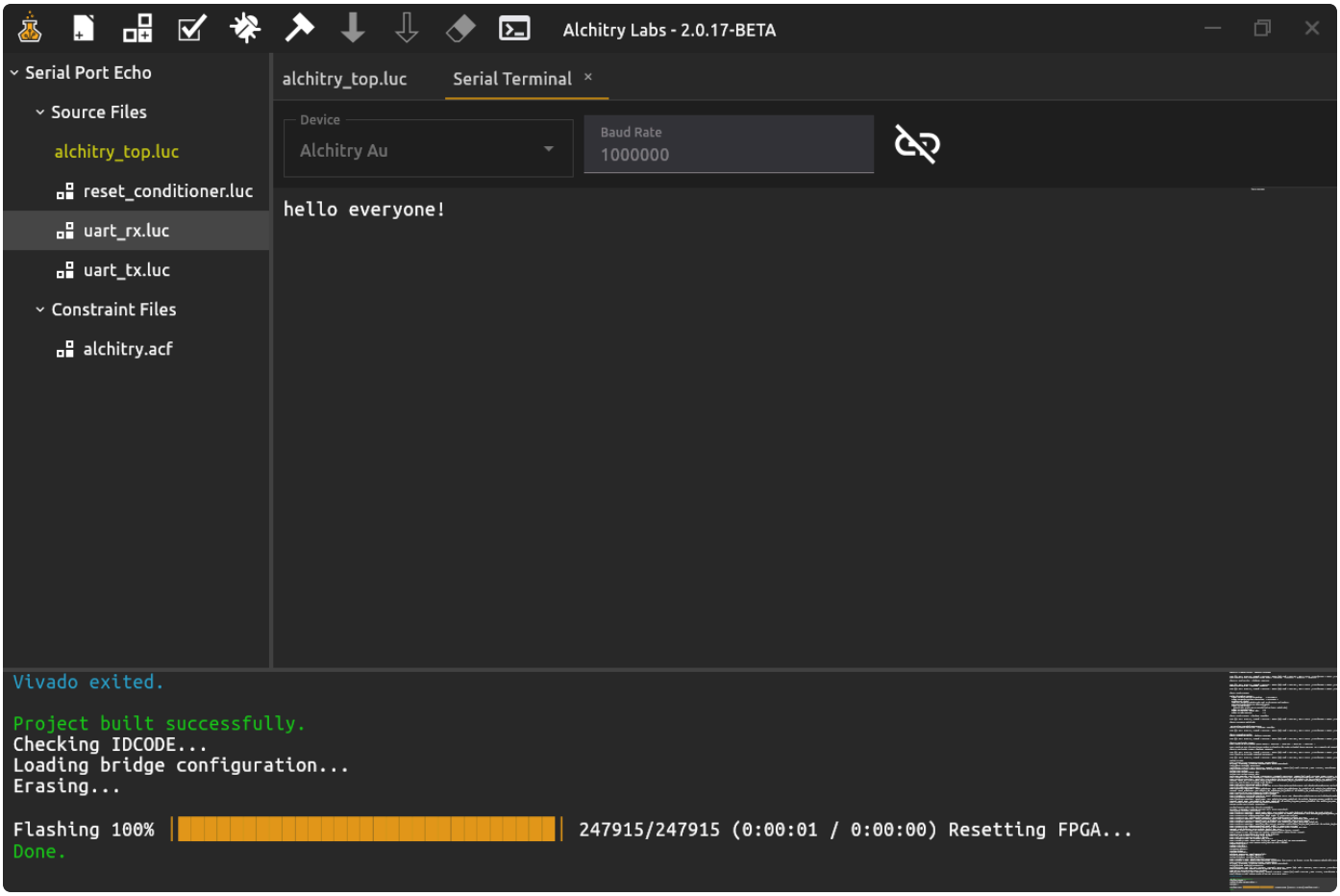
You should now be able to build your project and load it on the board.

Click the terminal icon in the toolbar and select *Serial Terminal* from the dropdown.

The *Device* dropdown shows all the detected boards. Make sure your board is plugged in and selected.

The default baud rate is 1M baud. The same as we set earlier so no need to change this.

Clicking the chain icon will connect to the board. You should then be able to type data into the monitor. Whatever you type should be shown.



Click the *break-chain* icon when you are done to disconnect.

Capturing Data

Before we wrap up, let's do a little more with the incoming data. We are going to save the last byte received and display it on the LEDs.

To do this, we need an 8 bit `dff`. The following line goes inside the `.clk(clk)` block but outside the `.rst(rst)` block. It could go inside the `.rst(rst)` block, but it really doesn't need a reset.

```
16  dff data[8]             // flip-flops to store last character
```

We can then write to the `dff` when we have new data. These lines go at the end of the `always` block. You can also remove the previous assignment of `0` to `led`.

```
35  if (rx.new_data) {      // new byte received
36      data.d = rx.data    // save it
37  }
38
39  led = data.q            // output the data
```

On the last line, we connect the LEDs to the output of the `dff`.

If you don't assign a `dff` a value, then it will retain the last value it had. Since we are only assigning it a value when `rx.new_data` is high, it will hold the last byte until the next one comes in.

Now, if you build and load your project, when you fire up a serial terminal you should not only see the text you send back in the monitor, but the LEDs on the board should also change depending on the character you last sent.

The full module looks like this.

```
module alchitry_top (
    input clk,              // 100MHz clock
    input rst_n,            // reset button (active low)
    output led[8],          // 8 user controllable LEDs
    input usb_rx,           // USB->Serial input
```

```
        output usb_tx           // USB->Serial output
    ) {

        sig rst                 // reset signal

        .clk(clk) {
            // The reset conditioner is used to synchronize the reset signal to
            // clock. This ensures the entire FPGA comes out of reset at the sa
            reset_conditioner reset_cond

            dff data[8]             // flip-flops to store last character

            .rst(rst) {
                uart_rx rx(#BAUD(1_000_000), #CLK_FREQ(100_000_000))
                uart_tx tx(#BAUD(1_000_000), #CLK_FREQ(100_000_000))
            }
        }

        always {
            reset_cond.in = ~rst_n  // input raw inverted reset signal
            rst = reset_cond.out    // conditioned reset

            rx.rx = usb_rx          // connect rx input
            usb_tx = tx.tx          // connect tx output

            tx.new_data = rx.new_data
            tx.data = rx.data
            tx.block = 0            // no flow control, do not block

            if (rx.new_data) {      // new byte received
                data.d = rx.data    // save it
            }

            led = data.q            // output the data
```

```
}
```

« PREV
Io Element

NEXT »
ROMs and FSMs

© 2025 Alchitry