



[Home](#) » [Tutorials](#) » [ROMs and FSMs](#)

# ROMs and FSMs

2024-09-19 · 14 min · Justin Rajewski | [Suggest Changes](#)

► [Table of Contents](#)

In this tutorial we will create a project that will send "Hello World!" over the USB (serial) port when the letter "h" is received. This will help teach you how to use finite state machines (FSM).

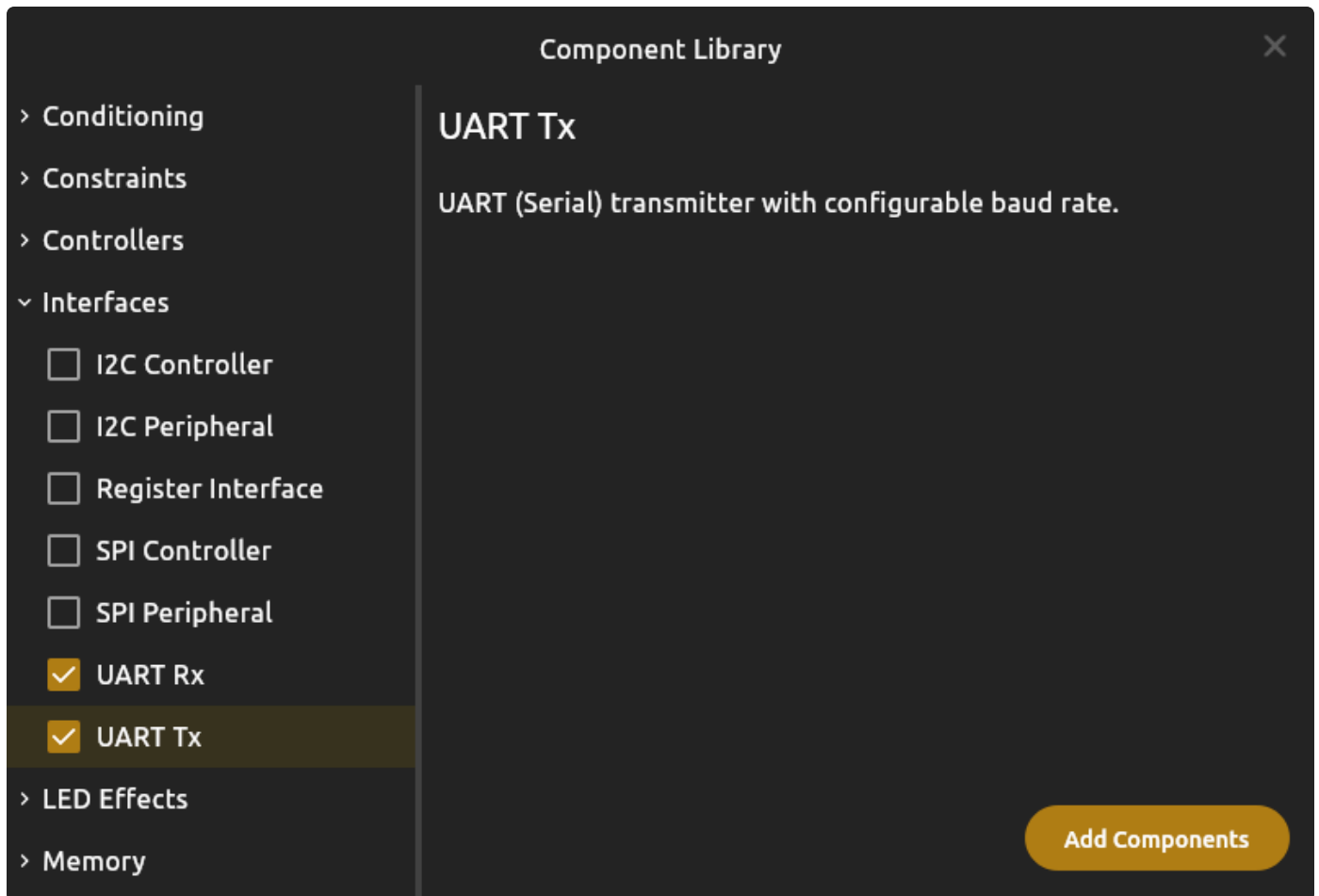
## Setup

We first need to create a new project based on the Base Project. I called mine *Hello World*, but you are free to choose whatever name you want.

With the new empty project, we now need to add the `uart_tx` and `uart_rx` components. These will be used to talk to the FTDI chip and send data over the USB port.

You should know how to add a component to your project from the last tutorial. If you need a refresher, [click here](#).

The components we need to add are the *UART Tx* and *UART Rx* components, and they can be found under *Interfaces*.



# UART Tx

Let's first take a look at what the module looks like.

```
module uart_tx #(
    CLK_FREQ ~ 100_000_000 : CLK_FREQ > 0,           // clock frequency
    BAUD ~ 1_000_000 : BAUD > 0 && BAUD <= CLK_FREQ/2 // desired baud rate
)(
    input clk,           // clock
    input rst,           // reset active high
    output tx,           // TX output
    input block,         // block transmissions
    output busy,         // module is busy when 1
    input data[8],       // data to send
```

```
input new_data    // flag for new data
```

We will only be looking at the interfaces to the modules since we don't need to know how it all works to use it properly (the magic of components).

This module is responsible for transmitting data. When we have a byte to send, we first need to check that `busy` is `0`. When this signal is `1`, any data we provide will be ignored. Assuming `busy` is `0`, we then provide the data to send on `data` and signal that the data is valid by setting `new_data` to `1`. This will cause the module to transmit the byte one bit at a time over `tx`.

The input `block` is used when you have some way of knowing that the device receiving the data upstream (the FTDI chip in this case) is busy. When `block` is `1`

can't hold more data, and it isn't a concern when the data is being read by an application on the PC side, we can set this permanently to `0`.

This module has two parameters you need to set in order to get it to work properly. The first one `CLK_FREQ` is simply the frequency of the clock you are providing it. If you are using the clock on the Alchitry board, this will be `100_000_000`, or 100MHz.

The second parameter, `BAUD` is used to set the rate of bits per second to send.

When a parameter is declared for a module, you only need to specify a name. However, you can also specify a default value and some value constraints.

The default value is set by using the equals sign.

You can also specify a test value instead of a default value by using `~` instead of `=`. A test value is used when checking the module for errors but will still force a value to be provided when the module is instantiated.

Typically, parameters are just numbers, but they can be more complex values like arrays. If you want to accept a multidimensional array, you need to specify the

dimensions with the default or test value. Parameters without a default or test value are assumed to be a simple number. See the [reference guide](#) for more info.

Constraints on the parameter's value can be set with a boolean statement after a colon. This expression will be evaluated when the module is instantiated and an error will be thrown when it fails (has a value of 0). It is recommended to add these constraints if you make any assumptions about the parameter values.

For both `CLK_FREQ` and `BAUD` it makes sense that they are not negative. For the module to work, the clock frequency needs to be at least twice the baud rate. Note that the closer you get to this limit, the more careful you need to be with choosing your baud rate. If the clock frequency isn't divisible by the baud rate then it will approximate the baud rate with the closest higher value.

## UART Rx

```
module uart_rx #(
    CLK_FREQ ~ 100_000_000 : CLK_FREQ > 0,           // clock frequency
    BAUD = 1_000_000 : BAUD > 0 && BAUD <= CLK_FREQ/4 // desired baud rate
)()
    input clk,           // clock input
    input rst,           // reset active high
    input rx,            // UART rx input
    output data[8],      // received data
    output new_data      // new data flag (1 = new data)
) {
```

This module is responsible for receiving data on the `rx` input and sending it out as bytes on `data`. The value of `data` is valid only when `new_data` is 1.

The parameters for this module are more or less the same as before with the small exception that `BAUD` is constrained to a quarter of `CLK_FREQ` instead of half. This is

due to the internal working of the module. Once it detects new incoming data, it waits half a cycle so that it will be sampling the data in the middle of the bit instead of the edge for reliability.

## Using the Modules

We now can add `uart_tx` and `uart_rx` to our top level module.

```

1  module alchitry_top (
2      input clk,                // 100MHz clock
3      input rst_n,              // reset button (active low)
4      output led[8],            // 8 user controllable LEDs
5      input usb_rx,             // USB->Serial input
6      output usb_tx             // USB->Serial output
7  ) {
8
9      sig rst                    // reset signal
10
11      .clk(clk) {
12          // The reset conditioner is used to synchronize the reset sig
13          // clock. This ensures the entire FPGA comes out of reset at
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          }
20      }
21
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 = 0           // no new data by default
32     tx.data = 8bx             // don't care when new_data is 0
33     tx.block = 0              // no flow control, do not block
34 }
```

All the external signals are already defined for us in the Base Project. We simply connect them up.

We can actually make the instantiation of these two modules a bit cleaner. Both `uart_tx` and `uart_rx` have the same parameters, and we want their values to be the same. This means we can group them in to a connection block just like we do for `clk` and `rst`.

```

11     .clk(clk) {
12         // The reset conditioner is used to synchronize the reset sig
13         // clock. This ensures the entire FPGA comes out of reset at
14         reset_conditioner reset_cond
15
16         .rst(rst) {
17             #BAUD(1_000_000), #CLK_FREQ(100_000_000) {
18                 uart_rx rx
19                 uart_tx tx
20             }
21         }
22     }
```

We could have also combined these with the `.rst(rst)` assignment, but we will be adding another module to that block later so it is nice to have them separate.

Currently, we are ignoring any data from the receiver and never sending data on the transmitter.

All inputs to modules need to be assigned a value. However, since we are setting `tx.new_data` to 0, we really don't care what value gets assigned to `tx.data` since it will never be used. In cases like this, the value of `bx` is helpful. There isn't really a value associated with `bx`. Instead, this tells the synthesizer that we don't care what value it uses. This gives it freedom to optimize our design instead of being forced to use an arbitrary useless value like `0`.

We will now create two new modules that will actually deal with all these signals to send "Hello World!" when an "h" is received.

## ROMs

Before we get too deep into generating and handling these signals, we need to create a ROM (Read **O**nly **M**emory).

Our ROM will hold the message we want to send, in our case "Hello World!".

Create a new module named `hello_world_rom` and add the following to it.

```

1  module hello_world_rom (
2      input address[4], // ROM address
3      output letter[8] // ROM output
4  ) {
5
6      const TEXT = "\r\n!dlrow olleH" // text is reversed to make 'H' a
7
8      always {
9          letter = TEXT[address] // address indexes 8 bit blocks of TEX
10     }
11 }
```

We have a single input, `address`, and a single output, `letter`. We want to output the first letter, `"H"`, when `address` is `0` and the second letter, `"e"`, when `address` is `1`. This continues for each letter in our message.

This is actually pretty simple to do. First we need an array of the data we want to send. This is done in the following line.

```
6  const TEXT = "\r\n!dlroW olleH" // text is reversed to make 'H' address
```

Here we are using a string to represent our data. Strings of more than one letter are 2D arrays. The first dimension has an index for each letter and the second dimension is 8 bits wide.

Note that the text is reversed. This is because we want, as the comment says, for `"H"` to be the first letter. Also note that `"\n"` and `"\r"` are actually single characters each. That means when we reversed the text we didn't write `"n\r"` which would be wrong. These characters will make sure the text is on a new line each time it is sent. `"\n"` goes to the next line and `"\r"` returns the cursor to the beginning of the new line.

Next, we simply need to set `letter` to the correct value in `TEXT` based on the given `address`. We do that on line 9.

```
9  letter = TEXT[address] // address indexes 8 bit blocks of TEXT
```

Since the text is reversed, we can simply output the corresponding letter.

This wraps up the ROM!

## The Greeter

This is where we will talk to the UART modules to actually send and receive data.



Create a new module named `greeter` and fill it with the following.

```
1  module greeter (  
2      input clk,          // clock  
3      input rst,          // reset  
4      input new_rx,       // new RX flag  
5      input rx_data[8],   // RX data  
6      output new_tx,      // new TX flag  
7      output tx_data[8],  // TX data  
8      input tx_busy       // TX is busy flag  
9  ) {  
10     const NUM_LETTERS = 14  
11  
12     enum States {IDLE, GREET}  
13  
14     .clk(clk) {  
15         .rst(rst) {  
16             dff state[$width(States)]  
17         }  
18         dff count[$clog2(NUM_LETTERS)] // min bits to store NUM_LETTERS  
19     }  
20  
21     hello_world_rom rom  
22  
23     always {  
24         rom.address = count.q  
25         tx_data = rom.letter  
26  
27         new_tx = 0 // default to 0  
28  
29         case (state.q) {  
30             States.IDLE:  
31                 count.d = 0  
32                 if (new_rx && rx_data == "h")  
33                     state.d = States.GREET  
34         }
```

```

35         States.GREET:
36             if (!tx_busy) {
37                 count.d = count.q + 1
38                 new_tx = 1
39                 if (count.q == NUM_LETTERS - 1)
40                     state.d = States.IDLE
41             }
42         }
43     }

```

The inputs and outputs should look a little familiar. They will connect to the `uart_tx` and `uart_rx` modules in our top level.

We are using the constant `NUM_LETTERS` to specify how big the ROM is. In our case, we have 14 letters to send (this includes the new line characters).

## Enums and FSMs

On line 12 we define an `enum`.

```

12 enum States {IDLE, GREET}

```

An `enum` is just a way to conveniently declare a group of constants that you don't particularly care about their values. You just need them to be different from each other.

We will be using this `enum` in conjunction with `dff` state to create an FSM (Finite State Machine).

Note that the width of `state` is defined as `$width(States)`. This ensures that `state` is wide enough to hold any value in `States`.

In this example, our FSM can have one of two states, `IDLE` or `GREET`. In a more complicated example we could add more states to our FSM simply by adding them to

the list.

To access a state, we can use `States.IDLE` or `States.GREET`. This is done in the case statement (covered below) as well as when we assign a new state to `state`.

## Functions

```
18 dff count[$clog2(NUM_LETTERS)] // min bits to store NUM_LETTERS - 1
```

Here we are declaring a counter that will be used to keep track of what letter we are on. That means we need the counter to be able to count from `0` to `NUM_LETTERS - 1`. How do we know how many bits we will need when `NUM_LETTERS` is a constant? We could simply compute this by hand and type in the value. However, this is fragile since it would be easy to change `NUM_LETTERS` and forget to change the counter size. This is where the function `$clog2()` comes in handy. This function will compute the ceiling log base 2 of the value passed to it. This happens to be the number of bits you need to store the values from `0` to one minus the argument. How convenient! Just what we needed.

You can check out the [reference guide](#) for more functions.

It is important to note that this function can only be used with constants or constant expressions. This is because the tools will compute the value during synthesis. Your circuit isn't doing anything fancy here. Computing this function in hardware would be far too complicated for a single line to properly handle.

## Saying Hello

We instantiate a copy of our `hello_world_rom` and call it `rom` so we know what data to send.

Since we are only going to be sending the letters from the ROM, we can wire them up directly to `tx_data`.

```
21 hello_world_rom rom
22
23 always {
24     rom.address = count.q
25     tx_data = rom.letter
```

We also can set the ROM's address to simply be the output of our counter since that's what the counter is for!

## Case Statements

**case** statements are an easy way to do a bunch of different things depending on the value of something. You could always use a bunch of **if** statements but this can be way more compact and easier to read.

The general syntax for a **case** statement is below.

```
case (expr) {
    const: statements
    const: statements
    const: statements
    default: statements
}
```

You pass in some expression and then have a bunch of blocks of statements that are considered based on the value of that expression. It sounds way more complicated than it is. Let's look at our example.

```
29 case (state.q) {
30     States.IDLE:
31         count.d = 0
32         if (new_rx && rx_data == "h")
33             state.d = States.GREET
34
```

```
35     States.GREET:
36         if (!tx_busy) {
37             count.d = count.q + 1
38             new_tx = 1
39             if (count.q == NUM_LETTERS - 1)
40                 state.d = States.IDLE
41         }
42     }
```

When `state.q` is `States.IDLE`, we only look at the lines 31-33. However, when `state.q` is `States.GREET` we only look at lines 36-41.

## Putting it all Together

So how does it all work? Since `IDLE` was the first state we listed, it is, by default, the default state. You can specify an alternate default state by using the parameter `#INIT(STATE_NAME)` on the `dff`.

Because we start in the idle state, the counter is set to `0`, and we do nothing until we see `"h"`. To wait for `"h"` we wait for `new_rx` to be high and `rx_data` to be `"h"`.

Once we receive an `"h"`, we change states to `States.GREET`.

Here we wait for `tx_busy` to be low to signal we can send data. We then increment the counter for next time and signal we have a new letter to send by setting `new_tx` high. Remember we already set `tx_data` as the output of our ROM.

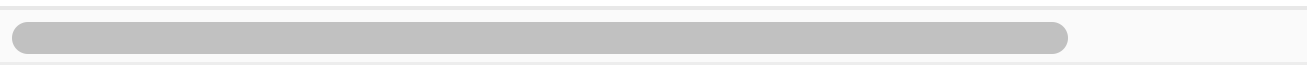
Once we are out of letters, we return to the idle state to wait for another `"h"`.

## Adding the Greeter to the Top Module

Finally, we need to add the `greeter` module to our top module.

First, let's add an instance of it.

```
11     .clk(clk) {
12         // The reset conditioner is used to synchronize the reset sig
13         // clock. This ensures the entire FPGA comes out of reset at
14         reset_conditioner reset_cond
15
16         .rst(rst) {
17             #BAUD(1_000_000), #CLK_FREQ(100_000_000) {
18                 uart_rx rx
19                 uart_tx tx
20             }
21
22             greeter greeter
23         }
24     }
```



Next, we need to connect it up.

```
26 always {
27     reset_cond.in = ~rst_n // input raw inverted reset signal
28     rst = reset_cond.out   // conditioned reset
29
30     led = 8h00             // turn LEDs off
31
32     rx.rx = usb_rx         // connect rx input
33     usb_tx = tx.tx        // connect tx output
34
35     greeter.new_rx = rx.new_data
36     greeter.rx_data = rx.data
37     tx.new_data = greeter.new_tx
38     tx.data = greeter.tx_data
```

```
39     greeter.tx_busy = tx.busy
40     tx.block = 0           // no flow control, do not block
41 }
```

That's it! Go ahead, build your project and load it on your board. You can then fire up a serial terminal and send `"h"` to your board to be nicely greeted!

[« PREV](#)[NEXT »](#)[Serial Interface](#)[Hello YOUR\\_NAME\\_HERE](#)

© 2025 [Alchitry](#)