

UART

Note Title

4/7/2010

Universal Asynchronous Receiver and Transmitter

This sends and receives parallel data over a serial line

It is the heart of the RS-232 standard,

It does use a different set of voltage levels than computer logic, so a voltage converter and interface is necessary.

* The transmitter is basically a parallel-in, serial out shift register.

* The receiver is basically a serial-in, parallel-out register

There is a bit sequence format that must be consistent on both transmitter and receiver ends.

* The four components of a data packet are:

(1) Start bit - Always a '0'

(2) Data bits - 6, 7, or 8 data bits

(3) Parity bit - Optional (for error detection),
Odd or Even parity

(4) Stop bit - Always a '1'; can be
1, 1.5, or 2 stop bits.

During idle times, the voltage on the line is held high (Logic '1')



The other parameter that needs to be known is the length of time to be used for each bit, i.e. how many bits per second or baud rate.

Common baud rates are 2400, 4800, 9600, 19.2k and further multiples.

UART Receiver

For the receiver you don't know when data is going to arrive or what the data bits are so we have to be able to know when data is arriving and when to read it

* The start bit signals the start of a data packet.

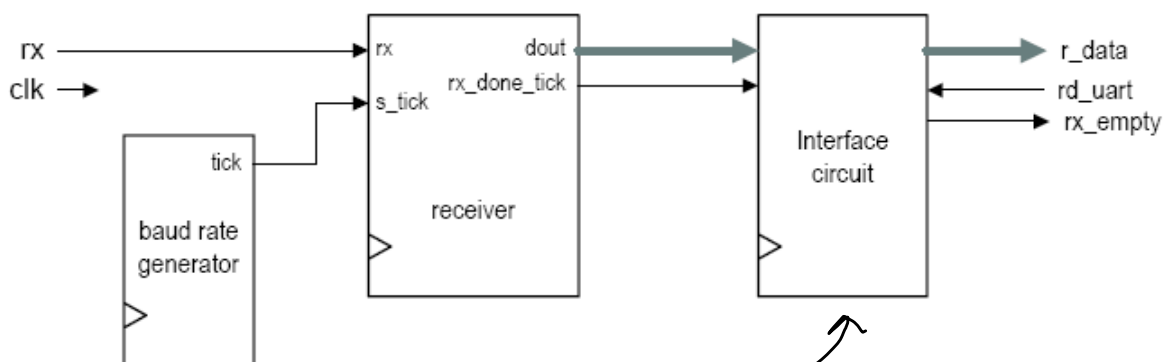
* Once we know it is starting, then we want to read the data line in the middle of each time slot.

* To get us "in the middle" we run a clock that is faster than the baud rate.

Common oversampling rate is 16 times faster

Over sampling Procedure:

- (1) Wait for RX line to go low (Logic '0').
Start the sampling tick counter
- (2) When the counter is at 7 we are halfway through the start bit. That is, we are now "in the middle", so reset the counter and start again
- (3) When the counter is equal to 15 we are in the middle of a data bit. Store that bit in our shift register and restart counter.
- (4) Repeat step 3 $N-1$ more times to store all N data bits.
- (5) If parity is being used, repeat step 3 to shift in the parity bit.
- (6) Repeat step 3 M times to shift in the M stop bits.



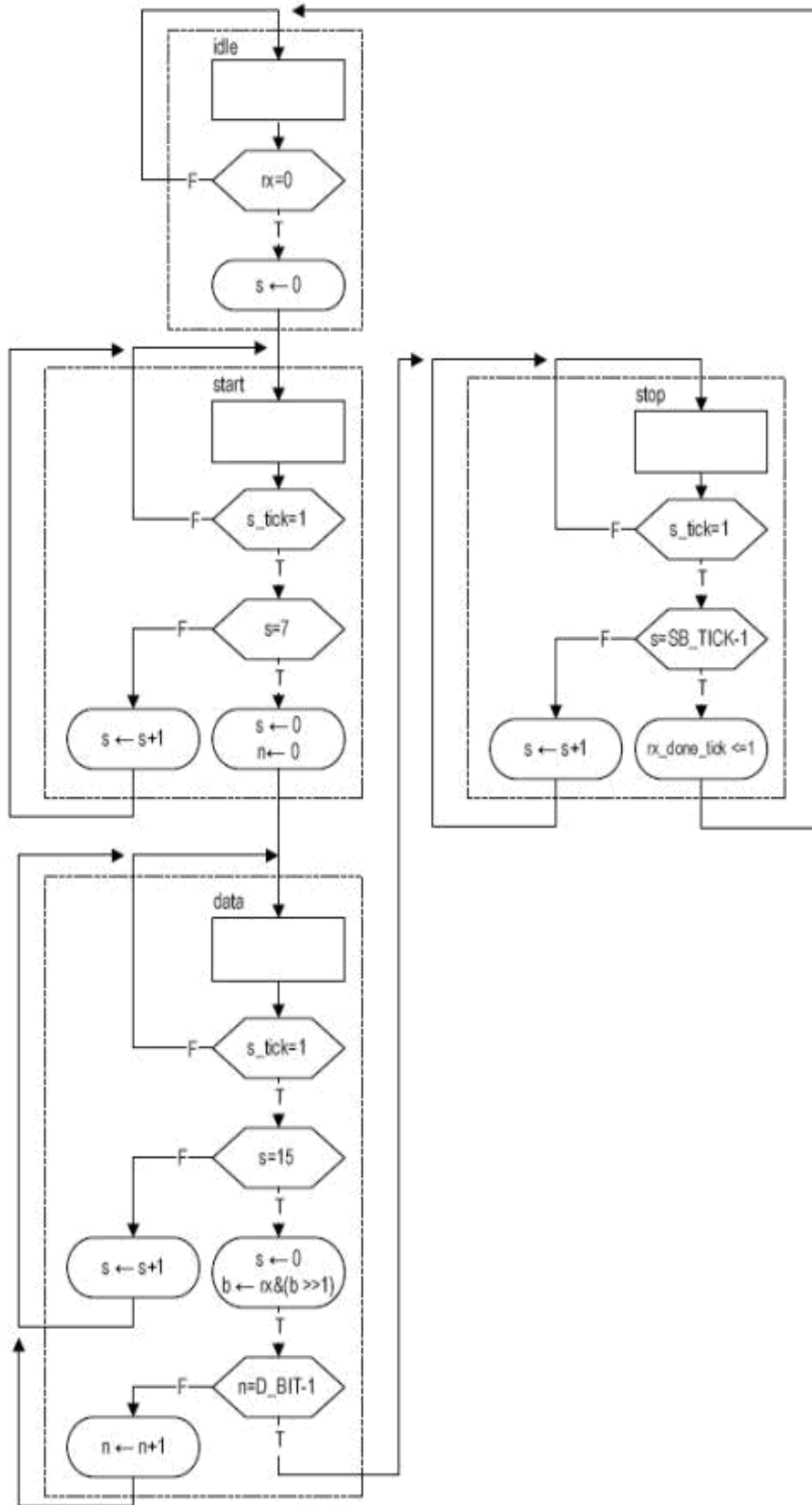
Need an interface to know when new data has arrived and/or been read.

Baud Rate Generator: If we want, for example, a 19200 baud rate we need an internal clock running 16 times faster (using our 50 MHz oscillator)

$$\frac{50,000,000}{19,200 \times 16} = 1 \text{ clock tick every } 162.76 \text{ oscillations.}$$

Thus, use a mod-163 counter as the book says.

Now let's take a look at the controller for the UART that coordinates the actions of the receiver.



D_BIT = the number of data bits being used

SB_TICK = the number of ticks needed to be in the middle of stop-bit

* No parity bit in this scheme

The n register keeps track of the number of data bits received.

The s register keeps track of the number of sampling ticks

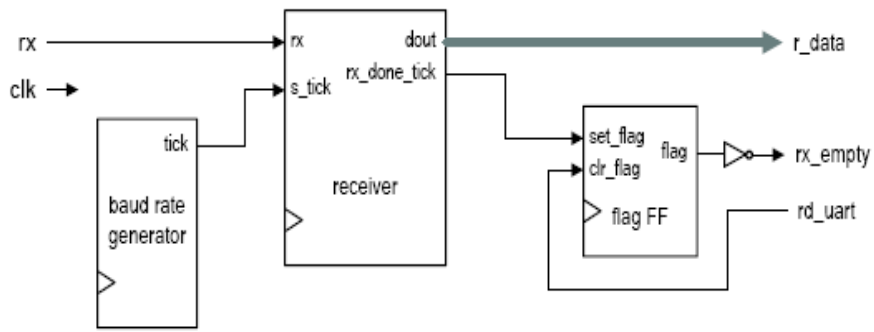
The b register holds the final word or packet.

To read out the data properly we need an interface circuit (for best operation).

This way, if the computer or device that is receiving the information is performing other tasks, it can complete those tasks and then get the info.

We need to let it know data is available.

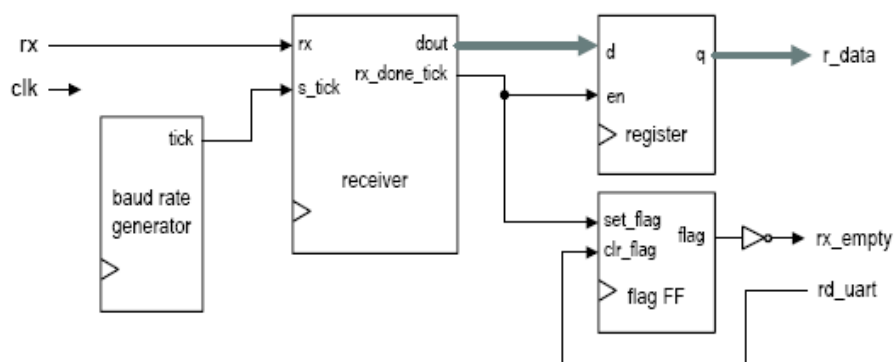
(1) Use a flag flip-flop.



(a) Flag FF

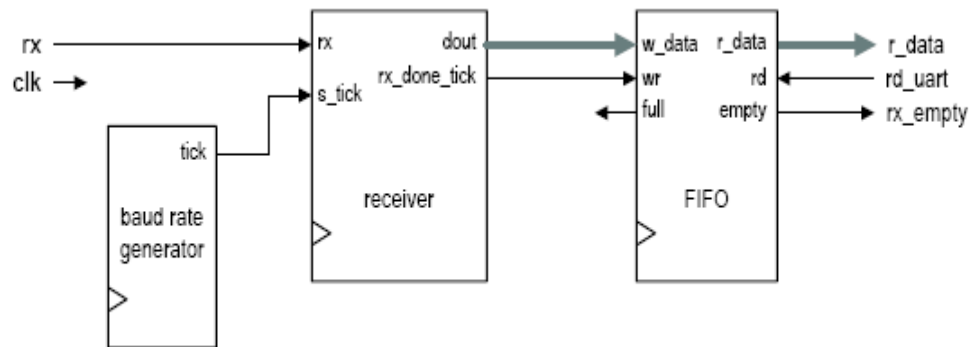
(2) Use a flag flip-flop and one-word buffer so next word can be received while waiting for system to retrieve first word.

If we don't retrieve the word in time we will have data overrun.



(b) Flag FF and one-word buffer

(3) Use a FIFO (First-In First-Out) Buffer to store several intermediate words while waiting to read them out.



(c) FIFO buffer

UART Transmitter

The operation of the UART transmitter is much like the operation of the receiver except that we don't have to guess where the middle of the bits are since the transmitter is in control.

See Listing 7.3

Now let's put the two parts together.

```

-- Listing 7.3
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_tx is
  generic(
    DBIT: integer:=8;      -- # data bits
    SB_TICK: integer:=16  -- # ticks for stop bits
  );
  port(
    clk, reset: in std_logic;
    tx_start: in std_logic;
    s_tick: in std_logic;
    din: in std_logic_vector(7 downto 0);
    tx_done_tick: out std_logic;
    tx: out std_logic
  );
end uart_tx ;

```

```

architecture arch of uart_tx is
  type state_type is (idle, start, data, stop);
  signal state_reg, state_next: state_type;
  signal s_reg, s_next: unsigned(3 downto 0);
  signal n_reg, n_next: unsigned(2 downto 0);
  signal b_reg, b_next: std_logic_vector(7 downto 0);
  signal tx_reg, tx_next: std_logic;
begin
  -- FSM state & data registers
  process(clk,reset)
  begin
    if reset='1' then
      state_reg <= idle;
      s_reg <= (others=>'0');
      n_reg <= (others=>'0');
      b_reg <= (others=>'0');
      tx_reg <= '1';
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
      s_reg <= s_next;
      n_reg <= n_next;
      b_reg <= b_next;
      tx_reg <= tx_next;
    end if;
  end process;
  -- next-state logic & data path functional units/routing
  process(state_reg,s_reg,n_reg,b_reg,s_tick,
    tx_reg,tx_start,din)
  begin
    state_next <= state_reg;
    s_next <= s_reg;
    n_next <= n_reg;
    b_next <= b_reg;
    tx_next <= tx_reg ;
    tx_done_tick <= '0';
    case state_reg is
      when idle =>
        tx_next <= '1';
        if tx_start='1' then
          state_next <= start;
          s_next <= (others=>'0');
          b_next <= din;
        end if;

```

3 registers hold similar information as the receiver

generator tick count
data bit count
bits to transmit

Hold to line high until time to transmit



```
when start =>
```

```
    if (s_tick = '1') then  
        if s_reg=15 then  
            state_next <= data;  
            s_next <= (others=>'0');  
            n_next <= (others=>'0');  
        else  
            s_next <= s_reg + 1;  
        end if;  
    end if;
```

```
when data =>
```

```
    tx_next <= b_reg(0);
```

```
    if s_reg=15 then  
        s_next <= (others=>'0');  
        b_next <= '0' & b_reg(7 downto 1);  
        if n_reg=(DBIT-1) then
```

```
            else  
                n_next <= n_reg + 1;  
            end if;
```

```
        else  
            s_next <= s_reg + 1;  
        end if;
```

```
when stop =>
```

```
    tx_next <= '1';  
    if (s_tick = '1') then  
        if s_reg=(SB_TICK-1) then  
            state_next <= idle;  
            tx_done_tick <= '1';  
        else
```

```
            s_next <= s_reg + 1;  
        end if;
```

```
    end if;
```

```
end case;
```

```
end process;
```

```
tx <= tx_reg;
```

```
end arch;
```

Get ready to drop TX
line low to indicate
start bit

Only move on to
data after start
bit low for 16 ticks

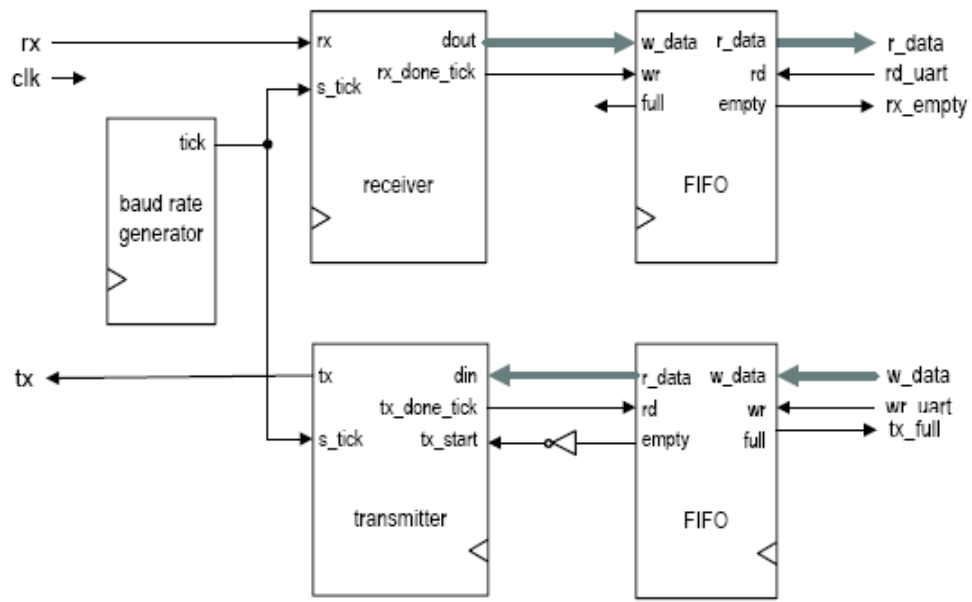
Put next data
bit on TX
line

Shift data
out once bit
has been
transmitted.

Go to stop bit
after all data

Complete UART

Communications
Link to
Another
CPU or
Device



CPU would read data out of FIFO with interrupts perhaps

CPU would fill FIFO with data to transmit

Figure 7.5 Block diagram of a complete UART.

See Listing 7.4

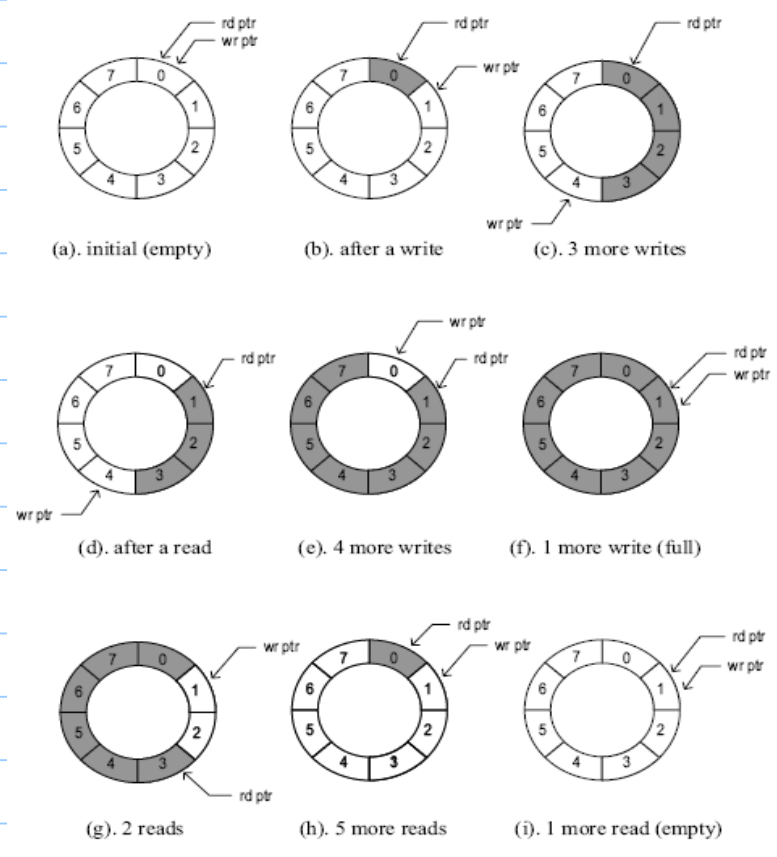


Figure 4.11 FIFO buffer based on a circular queue.

Testing Circuit

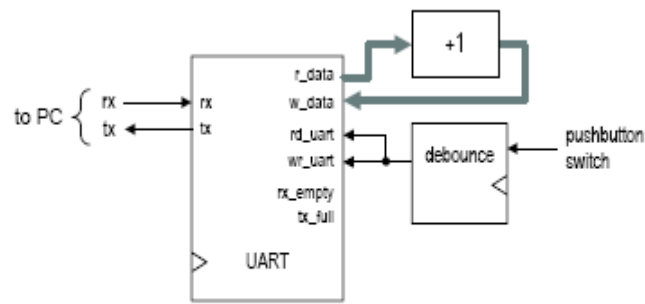


Figure 7.6 Block diagram of a UART verification circuit.

The receiver FIFO can fill up as we type characters on the PC.

We manually control pulling out one word at a time and looping it back to feed as the data to transmit back to the PC.

To make it obvious we are using data out of the UART, we increment the data by 1 so that it has obviously changed through the FPGA circuitry.

A debounced input switch will indicate that we want to read out from the receiver FIFO and write back to the transmitter FIFO

```
-- Listing 7.5
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_test is
  port(
    clk, reset: in std_logic;
    btn: std_logic_vector(2 downto 0);
    rx: in std_logic;
    tx: out std_logic;
    led: out std_logic_vector(7 downto 0);
    sseg: out std_logic_vector(7 downto 0);
    an: out std_logic_vector(3 downto 0)
  );
end uart_test;
```

```
architecture arch of uart_test is
  signal tx_full, rx_empty: std_logic;
  signal rec_data, rec_data1: std_logic_vector(7 downto 0);
  signal btn_tick: std_logic;
begin
  -- instantiate uart
  uart_unit: entity work.uart(str_arch)
    port map(
      clk=>clk, reset=>reset, rd_uart=>btn_tick,
      rx=>rx, tx=>tx, w_data=>rec_data,
      tx_full=>tx_full, rx_empty=>rx_empty,
      btn_tick=>btn_tick);
  -- instantiate debounce circuit
  btn_db_unit: entity work.btn_db(str_arch)
    port map(
      clk=>clk, reset=>reset, sw=>btn(0),
      btn_tick=>btn_tick);
  -- incremented data loop back
  rec_data1 <= std_logic_vector(unsigned(rec_data)+1);
  -- led display
  led <= rec_data;
  an <= "1110";
  sseg <= '1' & (not tx_full) & "11" & (not rx_empty) & "111";
end arch;
```

Connect up the UART with the feedback loop

Both a read and write are triggered by the button

Read out rec_data
Write back rec_data1

Increment value to feed back is always available and will be ready when button is pushed

Receiver data shows up on the LEDs

Full and Empty show up on 7-segment display

Hook up Button 0 to be debounced