Final Project
Serial Transmitter/Receiver
ECE 446

Peter Chinetti

December 2, 2014

Instructor: Professor Shanechi

Contents

1 Abstract

For this project, a simple serial receiver/transmitter pair was built. The pair operates at a selectable baud rate (1 or 10 baud). It was written in VHDL for implementation on the Spartan-3E FPGA board.

2 Implementation

The project is broken into two major components:

2.1 Transmitter

The transmitter is simple, a FSM looping between the states of idleness and transmission. The ports to the module are a 8 bit data vector, a ready line to signal the transmitter that the input vector has settled, a clock fed at the baud rate, an output to signal to the device feeding data transmission completed, and the transmission line.

From the idle state, after the input vector is loaded with data to be transmitted, the user must assert the ready signal. This prevents the transmission from starting before the correct data is loaded. After transmission has completed, the transmitter with assert its done signal, informing the user that new data can be loaded safely. The data on the input vector must remain stable between the time that the ready input is asserted and the done signal replies.

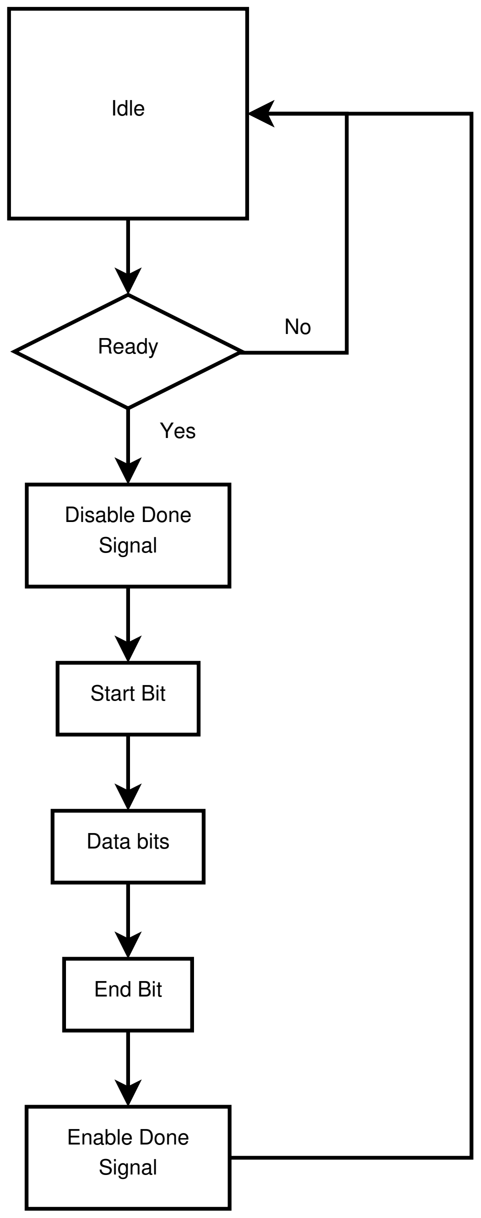
2.2 Receiver

The receiver is more complicated, to account for the asynchronous nature of the serial signal. It too is a state machine, but it has two 'clocks' the clk input is connected to a source running at 16 times the baud rate, and a resettable counter that divides that clock by 16. When the falling edge that starts the byte is sensed, the counter is aligned to sample at the center of each bit.

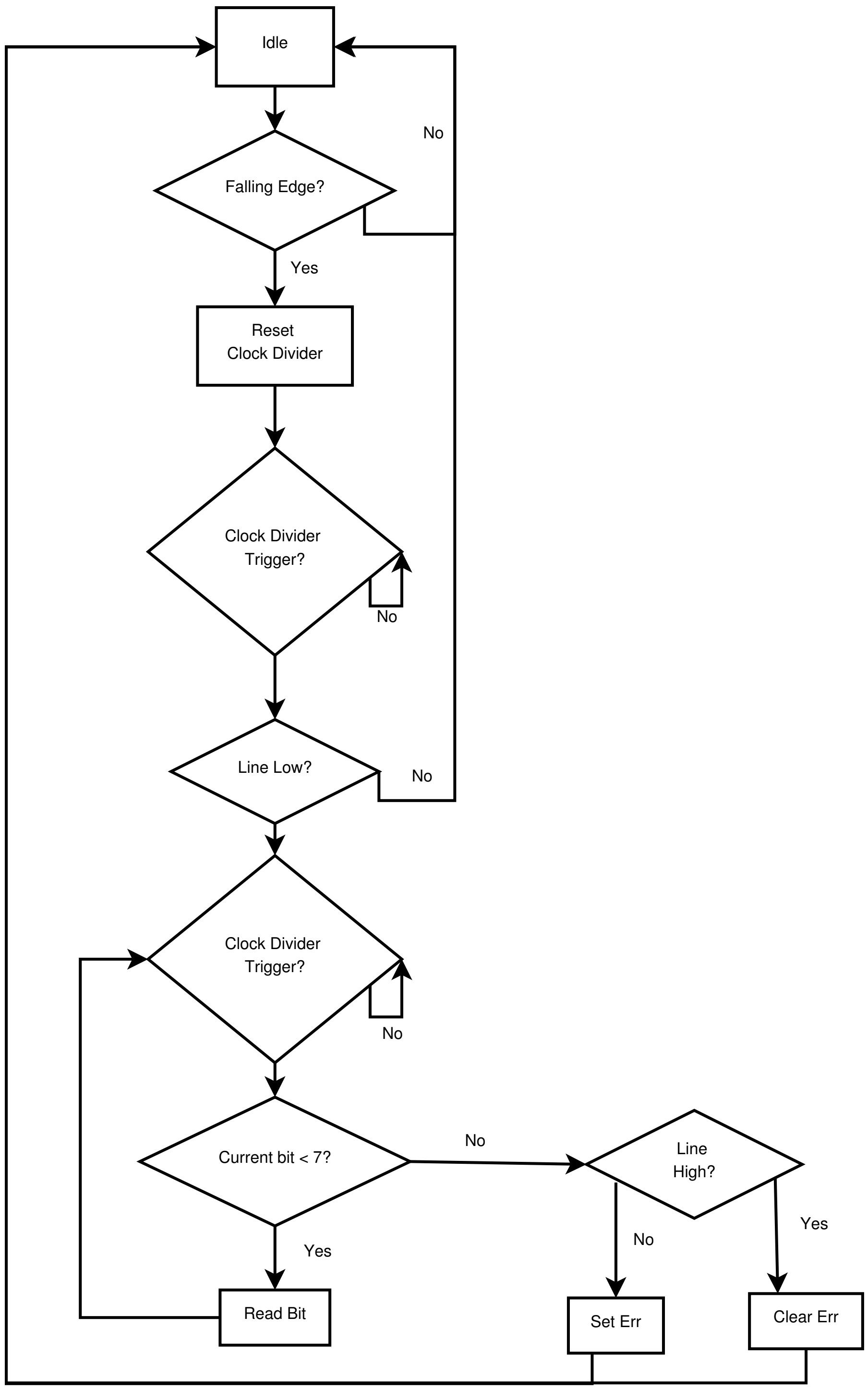
On the receiver, the done signal is asserted when the receiver is in an idle state, and the output vector can be safely sampled. If the done signal is not asserted, the vector is loading new data. The err signal is asserted if there is a framing error, informing the user to discard the output vector.

3 Flow Charts

3.1 Transmitter



3.2 Receiver



Idle
No
Falling Edge?
Yes
Reset Clock Divider
Clock Divider Trigger?
No
Line Low?
No
Clock Divider Trigger?
No
Current bit < 7?
No
Line High?
Yes
No
Yes
Read Bit
Set Err
Clear Err

4 Code Listings

4.1 Full Transmitter Receiver Pair

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity rx_tx is
  Port ( din : in  STD_LOGIC_VECTOR (7 downto 0);
         dout : out  STD_LOGIC_VECTOR (7 downto 0);
         speed : in  STD_LOGIC;
         clk: in STD_LOGIC;
         strataflash_oe : out std_logic;
         strataflash_ce : out std_logic;
         strataflash_we : out std_logic;
         lcd_d : inout std_logic_vector(7 downto 4);
         lcd_rs : out std_logic;
         lcd_rw : out std_logic;
         lcd_e : out std_logic);
end rx_tx;

architecture Behavioral of rx_tx is
  component rx is
    Port ( rx_line : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           err : out  STD_LOGIC := '1';
           data : out  STD_LOGIC_VECTOR (7 downto 0) := "00000000";
           ready : out  STD_LOGIC);
  end component;
  component tx is
    Port ( clk : in  STD_LOGIC;
           ready: in STD_LOGIC;
           data : in  STD_LOGIC_VECTOR (7 downto 0);
           tx_line : out STD_LOGIC;
           done : out  STD_LOGIC);
  end component;
  component LCD_IF is
    Port (      int1         : in std_logic_vector(3 downto 0);
           int2         : in std_logic_vector(3 downto 0);
           int3         : in std_logic_vector(3 downto 0);
           int4         : in std_logic_vector(3 downto 0);
           strataflash_oe : out std_logic;
           strataflash_ce : out std_logic;
           strataflash_we : out std_logic;
           lcd_d : inout std_logic_vector(7 downto 4);
           lcd_rs : out std_logic;
           lcd_rw : out std_logic;
           lcd_e : out std_logic;
           clk_50mhz : in std_logic);
  end component;
  --The selectable clocks feed the data rates required for 1 and 10
     baud comms
  --as well as the 16x multiplied clock the rx side needs.
  component selectable_clock is
    Port ( clk : in std_logic;
           s0 : in std_logic;
```

```
          s1 : in std_logic;
          out_clk : out std_logic);
  end component;
  signal input_buffer, output_buffer : STD_LOGIC_VECTOR(7 downto 0)
    ;
  signal rx_sel_0, rx_sel_1, tx_sel_0, tx_sel_1 : STD_LOGIC;
  signal rx_clk, tx_clk : STD_LOGIC;
  signal tx_ready, tx_done : STD_LOGIC;
  signal comms_line : STD_LOGIC;
  signal rx_ready, rx_err : STD_LOGIC;

begin
  tx_0 : tx
  port map(
      clk => tx_clk,
      ready => tx_ready,
      data => input_buffer,
      tx_line => comms_line,
      done => tx_done);
  rx_0 : rx
  port map(
      rx_line => comms_line,
      clk => rx_clk,
      err => rx_err,
      data => output_buffer,
      ready => rx_ready);
  rx_clk_sel : selectable_clock
  port map(
      clk => clk,
      s0 => rx_sel_0,
      s1 => rx_sel_1,
      out_clk => rx_clk);
  tx_clk_sel : selectable_clock
  port map(
      clk => clk,
      s0 => tx_sel_0,
      s1 => tx_sel_1,
      out_clk => tx_clk);
  lcd_0 : LCD_IF
  port map(
      int1 => input_buffer(3 downto 0),
      int2 => input_buffer(7 downto 4),
      int3 => output_buffer(3 downto 0),
      int4 => output_buffer(7 downto 4),
      strataflash_oe => strataflash_oe,
      strataflash_ce => strataflash_ce,
      strataflash_we => strataflash_we,
      lcd_d => lcd_d,
      lcd_rs => lcd_rs,
      lcd_rw => lcd_rw,
      lcd_e => lcd_e,
      clk_50mhz => clk);
  --locking the tx ready line ready to transmit
  tx_ready <= '1';
  process(tx_done, din)
  begin
    --using the 'done' line for flow control instead
    if tx_done = '1' then
      input_buffer <= din;
    end if;
  end process;
  process(rx_ready, rx_err, output_buffer)
  begin
    --this might be hard to debug, as errors are opaque
    if rx_ready = '1' and rx_err = '0' then
      dout <= output_buffer;
    end if;
  end process;
  process(speed)
  begin
    if speed = '1' then
      --10 baud rx/tx
      --rx runs at 160 Hz
      rx_sel_0 <= '1';
      rx_sel_1 <= '1';
      --tx runs at 10 Hz
      tx_sel_0 <= '0';
      tx_sel_1 <= '1';
    else
      --1 baud rx/tx
      --rx runs at 16 Hz
      rx_sel_0 <= '0';
      rx_sel_1 <= '0';
      --tx runs at 1 Hz
      tx_sel_0 <= '1';
      tx_sel_1 <= '0';
    end if;
  end process;
end Behavioral;
```

../RS–232/rx_tx.vhd

4.2 Transmitter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity tx is
    Port ( clk : in  STD_LOGIC;
     ready: in STD_LOGIC;
           data : in  STD_LOGIC_VECTOR (7 downto 0);
     tx_line : out STD_LOGIC;
           done : out  STD_LOGIC);
end tx;

architecture Behavioral of tx is
type state is (idle, start_bit, bit_0, bit_1, bit_2, bit_3, bit_4,
    bit_5, bit_6, bit_7, stop_bit);
signal f : state;
signal p : state := idle;

```

```
begin
process (clk)
begin
  -- FSM based implementation switches at the clocks
  if rising_edge(clk) then
    p <= f;
  end if;
end process;

process (p,ready, data)
begin
  case p is
    when idle =>
      --The ready line unlocks the FSM
      --from it's idle state. It exists
      --to allow the input lines to settle
      tx_line <= '1';
      done <= '1';
      if ready = '1' then
        f <= start_bit;
      else
        f <= idle;
      end if;
    when start_bit =>
      done <= '0';
      tx_line <= '0';
      f <= bit_0;
    when bit_0 =>
      tx_line <= data(7);
      f <= bit_1;
    when bit_1 =>
      tx_line <= data(6);
      f <= bit_2;
    when bit_2 =>
      tx_line <= data(5);
      f <= bit_3;
    when bit_3 =>
      tx_line <= data(4);
      f <= bit_4;
    when bit_4 =>
      tx_line <= data(3);
      f <= bit_5;
    when bit_5 =>
      tx_line <= data(2);
      f <= bit_6;
    when bit_6 =>
      tx_line <= data(1);
      f <= bit_7;
    when bit_7 =>
      tx_line <= data(0);
      f <= stop_bit;
    when stop_bit =>
      tx_line <= '1';
      f <= idle;
  end case;
end process;
end Behavioral;
```

../RS–232/tx.vhd

4.3 Receiver

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity rx is
  Port ( rx_line : in  STD_LOGIC;
         clk : in  STD_LOGIC;
         err : out  STD_LOGIC := '1';
         data : out  STD_LOGIC_VECTOR (7 downto 0) := "00000000";
         ready : out  STD_LOGIC);
end rx;

architecture Behavioral of rx is
  component clk_sampler is
    Port ( reset : in  STD_LOGIC;
           clk :in STD_LOGIC;
           full : out  STD_LOGIC);
  end component;
  signal sampler_full, sampler_reset : STD_LOGIC;
  type state is (idle, start_bit, bit_0, bit_1, bit_2, bit_3, bit_4
    , bit_5, bit_6, bit_7, stop_bit);
  signal f : state;
  signal p : state := idle;
begin
  cs_0 : clk_sampler
  port map(
      clk => clk,
      reset => sampler_reset,
      full => sampler_full);
  process (clk)
  begin

    if rising_edge(clk) then
      p <= f;
    end if;
  end process;
  process(rx_line, sampler_full, p)
  begin
    case p is
      when idle =>
        if rx_line = '0' then
          --falling edge of start bit detected,
          --reset the clock divider to 1/2 period
          sampler_reset <= '1';
          f <= start_bit;
        end if;
        ready <= '1';
      when start_bit =>
        ready <= '0';
```

```
          sampler_reset <= '0';
          if sampler_full = '1' then
            if rx_line = '0' then
              f <= bit_0;
            else
              f <= idle;
            end if;
          end if;
        when bit_0 =>
          data(7) <= rx_line;
          if sampler_full = '1' then
            f <= bit_1;
          else
            f <= bit_0;
          end if;
        when bit_1 =>
          data(6) <= rx_line;
          if sampler_full = '1' then
            f <= bit_2;
          else
            f <= bit_1;
          end if;
        when bit_2 =>
          data(5) <= rx_line;
          if sampler_full = '1' then
            f <= bit_3;
          else
            f <= bit_2;
          end if;
        when bit_3 =>
          data(4) <= rx_line;
          if sampler_full = '1' then
            f <= bit_4;
          else
            f <= bit_3;
          end if;
        when bit_4 =>
          data(3) <= rx_line;
          if sampler_full = '1' then
            f <= bit_5;
          else
            f <= bit_4;
          end if;
        when bit_5 =>
          data(2) <= rx_line;
          if sampler_full = '1' then
            f <= bit_6;
          else
            f <= bit_5;
          end if;
        when bit_6 =>
          data(1) <= rx_line;
          if sampler_full = '1' then
            f <= bit_7;
          else
            f <= bit_6;
          end if;
```

```
      when bit_7 =>
        data(0) <= rx_line;
        if sampler_full = '1' then
          f <= stop_bit;
        else
          f <= bit_7;
        end if;
      when stop_bit =>
        if sampler_full = '1' then
          if rx_line = '1' then
            err <= '0';
          else
            err <= '1';
          end if;
          f <= idle;
        else
          f <= stop_bit;
        end if;
    end case;

  end process;


end Behavioral;
```

../RS–232/rx.vhd

4.3.1 Receiver Clock Divider

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;


entity clk_sampler is
  Port ( reset : in  STD_LOGIC;
         clk :in STD_LOGIC;
         full : out  STD_LOGIC);
end clk_sampler;

architecture Behavioral of clk_sampler is
  signal counter: unsigned(3 downto 0) :="0000";
begin

  process(clk, reset)
  begin
    if reset = '1' then
      --reset for a half count.
      counter <= "1000";
    elsif rising_edge(clk) then
      counter <= counter + "0001";
    end if;
  end process;

  process(counter)
  begin
    --Full is 1/16 of the incoming frequency
    if counter = "1111" then
      full <= '1';
    else
      full <= '0';
    end if;
  end process;

  process(reset)
  begin
  end process;

end Behavioral;
```

../RS–232/clk_sampler.vhd

4.4 Clock Divider

```
-- Selectable output frequency clock divider code.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity selectable_clock is
  Port ( clk : in std_logic;
         s0 : in std_logic;
         s1 : in std_logic;
         out_clk : out std_logic);
end selectable_clock;
--If s1 and s0 are both low, the output clock rate is 16 Hz.
--If s1 is low and s0 is high, the output clock rate is 1 Hz.
--If s1 is high and s0 is low, the output clock rate is 10 Hz.
--If s1 and s0 are both high, the output clock rate is 160 KHz.
architecture Behavioral of selectable_clock is
begin
  process (clk, s0, s1)
    variable count : integer := 0;
  begin
    if clk = '1' and clk'event then
      count := count + 1;
      -- Start a process.
      -- Variable declaration.
      -- Rising edge detection.
      -- Code to create the 16 Hz clock.
      if s0 = '0' and s1 = '0' then
        if count >= 3125000 then
          -- Taken off a 50MHz clock.
          count := 0;
        -- Reset count for next cycle.
        end if;
        if count >= 0 and count <= 1562500 then
          out_clk <= '1';
        -- High portion of 16 HZ clock.
        else
```

```
          out_clk <= '0';
        -- Low portion of 16 HZ clock.
        end if;
      end if;
      -- Code to create the 1 Hz clock.
      if s0 = '1' and s1 = '0' then
        if count >= 50000000 then
          -- Taken off a 50MHz clock.
          count := 0;
        -- Reset count for next cycle.
        end if;
        if count >= 0 and count <= 25000000 then
          out_clk <= '1';
        -- High portion of 1 HZ clock.
        else
          out_clk <= '0';
        -- Low portion of 1 HZ clock.
        end if;
      end if;
      -- Code to create the 10 Hz clock.
      if s0 = '0' and s1 = '1' then
        if count >= 5000000 then
          -- Taken off a 50MHz clock.
          count := 0;
        -- Reset count for next cycle.
        end if;
        if count >= 0 and count <= 2500000 then
          out_clk <= '1';
        -- High portion of 10 HZ clock.
        else
          out_clk <= '0';
        -- Low portion of 10 HZ clock.
        end if;
      end if;
      -- Code to create the 160 Hz clock.
      if s0 = '1' and s1 = '1' then
        if count >= 312500 then
          -- Taken off a 50MHz clock.
          count := 0;
        -- Reset count for next cycle.
        end if;
        if count >= 0 and count <= 156250 then
          out_clk <= '1';
        -- High portion of 160 Hz clock.
        else
          out_clk <= '0';
        -- Low portion of 160 Hz clock.
        end if;
      end if;
    end if;
  end process;
end Behavioral;
```

../RS–232/clk_divider.vhd

5 Implementation Constraints

The only constraints in this project were the desired baud rate and the available space on the FPGA. This project had neither the speed nor the size to pose a serious implementation problem.

6 Engineering and Design Challenges

The most challenging section of this design was manipulating the 50MHz onboard clock to produce two sets of three slower rates. The selectable clock divider circuit provided in the lab manuals was modified to produce the required rates.

The rest of the design was a straightforward series of FSMs.

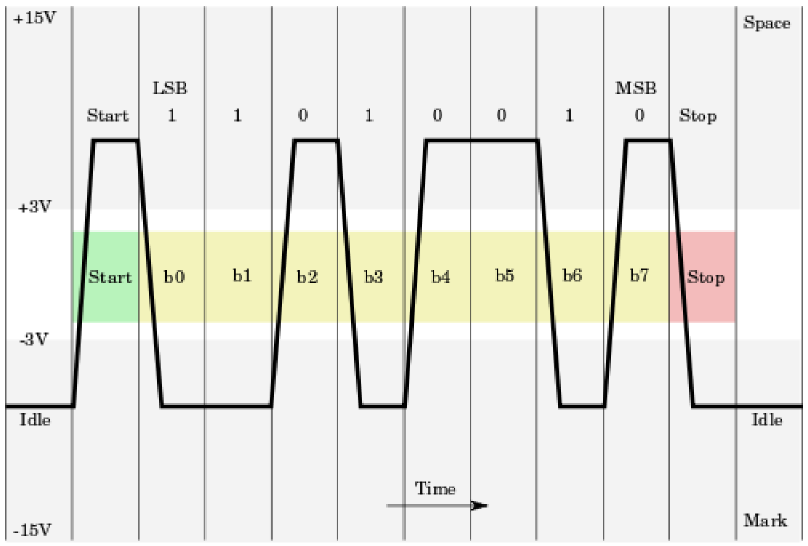
7 Possibilities for Expansion

This asynchronous communication unit could be used in any 3.3 volt digital circuit that required only a low speed link, as the data transfer rate is very low. A faster version of the circuit is used to interconnect many hobbyist electronics project, such as those using an Arduino.

If the design requirements were changed to include a need for error correction, hamming codes could be interwoven into the data bits to provide that functionality. Unfortunately, these bits would slow data rate, and each frame would either have to be extended or it would transmit fewer bits.

8 Alternative Techniques

What we have implemented in this lab is not true RS-232, as RS-232 has voltages between -15 to -3 and 3 to 15 volts, not 0 to 3.3 as this transmitter actually outputs. A diagram of an RS-232 transmission of the letter 'K' is below:



[2]

Instead, what we built in lab is an UART (Universal Asynchronous Receiver/-Transmitter). The UART protocol has the high idle states and non-reversed signaling at DC voltages. [3]

A limitation both RS-232 and the implemented transmitter is data rate. It is very difficult to increase the data rate of these designs enough to make them competitive with their successor interconnect protocol: USB. USB transmissions happen on a pair of differential lines, where the signal is protected somewhat from noise. This and other improvements allow USB to transmit at 480 Mbit/s, significantly higher than is possible with RS-232. [1]

9 Conclusion

Much was learned in the construction of this unit. Integration and testing of small modules was successfully performed. Working with pre-built modules such as the LCD driver unit was also practiced. The unit works, and could be useful as a component in low speed applications.

References

[1] Compaq, HP, Intel, Lucent, Microsoft, NEC and Philips, *Universal Serial Bus Specification Revison 2.0.* USB Implementers Forum, 2000

[2] Samuel Tardieu, *Rs232 Oscilloscope trace*, Web, March, 2009

[3] Various, *Universal Asynchronous Receiver/Transmitter* Wikipedia, 2014