Engs 31 / CoSc 56

Lab 5: Serial Communications

Due in lab, week of May 9

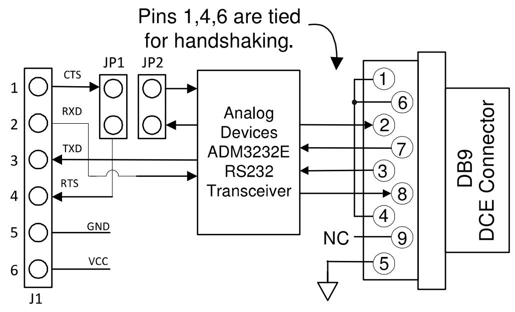
Learning objective

1. Understand how asynchronous serial communication works.
2. Continue to develop skill with modular design, construction, and testing.
3. Continue to develop skill with the oscilloscope.

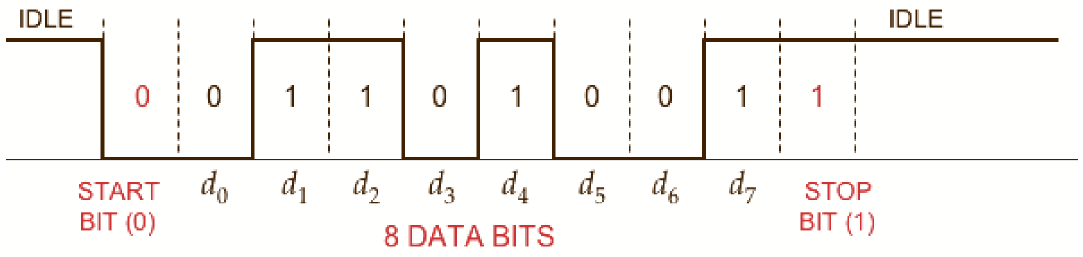
Introduction

The Digilent PmodRS232X supplies an RS-232 serial port expansion for the Nexys3 board, as shown below. In the jargon of the RS-232 convention, the PmodRS232X is a DCE device, and a PC is a DTE device. A DCE and DTE may be connected together via a "straight through" cable having a male DB-9 connector on one end and a female DB-9 connector on the other end. This cable connects the receive and transmit wires (RXD and TXD, respectively) of the PmodRS232X to the TXD and RXD wires of the PC's serial port, enabling two-way communication between the PC and the Nexys3.

Conversion between RS-232 voltage levels (-3 to -12 volts for logical 1 and +3 to +12 volts for logical 0) and normal high-true logic levels is taken care of by a voltage converter chip on the board. As seen in the diagram, the transmit and receive signals are connected to pins 2 and 3 of the Pmod connector. Transmit and receive ports in the logic you design in this lab will be mapped to these pins, and hence to the RS-232 port, via the UCF file.

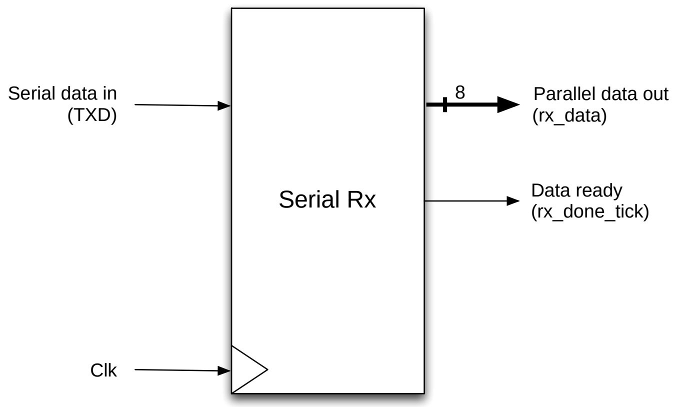

Pins 1,4,6 are tied for handshaking.
JP1
JP2
CTS
RXD
TXD
RTS
GND
VCC
1
2
3
4
5
6
J1
Analog Devices ADM3232E RS232 Transceiver
NC
DB9 DCE Connector

In the RS-232 asynchronous serial communications protocol, an 8-bit byte (typically an ASCII-encoded character) is transmitted as shown in the picture below. When nothing is being sent (the line is quiescent), the signal is at a logical "1" level. The byte is preceded by a "0", called the *start bit*, and concluded by a "1", called the *stop bit*. A parity bit may optionally be inserted between the most significant data bit and the stop bit, and there may be an additional stop bit. But this packet format, "8 bits, no parity, 1 stop bit", is common and the one we shall use in the lab.

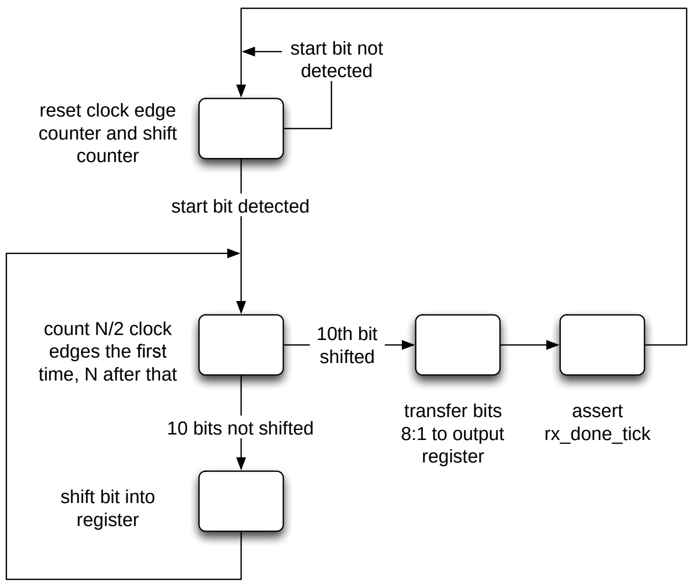


The reciprocal of the temporal width of a single bit is called the *baud rate*. Common baud rates are 9600 and 19,200. The transmitter is simply a shift register that takes in the 8-bit data, adds a start bit and stop bit, and shifts the result out serially at the baud rate. The data packets are transmitted without an accompanying clock (hence, "asynchronous"), so the clock must be recovered in the receiver.

Your task in this lab is to build the receiver. It is assumed that the receiver knows the transmitter's baud rate. The receiver's inputs are the serial data stream and a 10 MHz master clock. Its outputs are the eight-bit data byte, in parallel, and a data ready signal that pulses high for one master clock cycle (*i.e.*, 100 ns) when all the bits in the packet have been received and the data byte is available at the output.



We may surmise that, at least, the receiver consists of a 10-bit shift register and a state machine to control the shift register. The baud rate generator is shown as external to the receiver block, because it is also used by the transmitter block. The operation of the receiver is described by the following high-level state machine.


start bit not detected
reset clock edge counter and shift counter
start bit detected
count N/2 clock edges the first time, N after that
10th bit shifted
transfer bits 8:1 to output register
assert rx_done_tick
10 bits not shifted
shift bit into register

- Watch the serial input line, waiting for it to drop from 1 to 0, signifying the beginning of a packet.

- Find the center of the start bit. The number of clock edges in one bit time is

  N = bit time / clock period = clock frequency / baud rate

Counting $N/2$ clock edges after the serial input line drops identifies the center of the start bit. At this point, shift the register, bringing the start bit into the register's msb.

- Count N more clock edges, which locates the center of the first data bit. Shift this value into the register. Repeat this step until the stop bit is shifted into the register (count 10 shifts).
- After 10 shifts, the middle eight bits of the register (8 down to 1) contain the data byte. Transfer this byte into an output register. This second register ensures that the output doesn't fluctuate while the next packet is being received.
- Assert the rx_done_tick pulse for one clock cycle to signal that the byte is available, then return to the initial state and wait for the next start bit.

In addition to the shift register, we will need an 8-bit parallel in/out register and two counters, one to count clock edges and one to count how many bits have been shifted in. You will also need the multiplexed seven-segment display component from Lab 4.

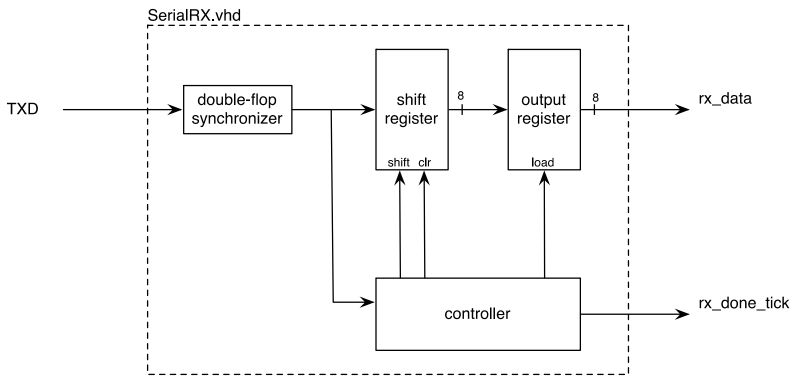
BEFORE COMING TO LAB

Reading

- Digilent *PmodRS232X Reference Manual* (Canvas)
- There's a table of ASCII codes at http://www.asciitable.com/
- Agilent *InfiniiVision 7000B Series Oscilloscope User Guide*, Chapter 15 (pp 333-342), "UART/RS232 Triggering and Serial Decode" (Canvas)

Part I: Prelab

The block diagram of the receiver is shown below. All blocks are clocked by the 10 MHz master clock (not shown, for simplicity).



Put all the blocks in one VHDL model, called SerialRX.vhd, and use the following entity:

```
entity SerialRx is
   Port ( Clk : in  STD_LOGIC;                          -- 10MHz master clock
          RsRx  : in  STD_LOGIC;                        -- received bit stream
          rx_shift : out STD_LOGIC;                     -- for testing
```

```
            rx_data : out  STD_LOGIC_VECTOR (7 downto 0);  -- data byte
            rx_done_tick : out  STD_LOGIC );               -- data ready tick
   end SerialRx;
```

Proceed by the following steps:

1. *Synchronizer*
The asynchronous TXD signal from the Pmod-RS232X must be synchronized with the 10 MHz master clock. Design a **two flip-flop synchronizer** (see Vahid, pp 147-149, metastability) as a process in your model.

2. *Shift register and data register*
Design a **10-bit shift register**. It will have a CLEAR input for synchronous resetting, and a SHIFT input that enables the register to shift right on a clock edge. The serial input is through bit 9, the msb. It will be clocked with the 10 MHz master clock.

Add an **8-bit parallel-load register** that copies bits 8 down to 1 of the shift register when enabled by a control signal. With this register the receiver output, rx_data, will remain stable while the next byte is being received into the shift register.

3. *Controller*
The controller consists of a finite state machine and two **counters**:

- a counter to count $N/2$ clock edges before the first shift and N clock edges between shifts (Lab 4 postlab)
- a counter to count ten bits shifted into the register

They both require synchronous initialization.

Design the **controller FSM**. Its inputs are the synchronized serial data stream (so you can watch for the start bit) and the terminal count signals from the two counters. The controller outputs are control signals for the shift register, the data register, and the two counters, plus the data ready signal rx_done_tick.

For debugging in the lab, plan to bring the input data stream, the 10 MHz clock, the shift register's SHIFT signal (I call it rx_shift), and rx_done_tick out to unused Pmod pins on the FPGA board. These signals are invaluable in helping debug your design.

In the lab, the serial input to the Pmod-RS232X will come from a PC running an application called *PuTTy*, and the 8 output bits from the receiver will be sent two places: to the multiplexed seven-segment display and to the serial transmitter for looping back up the RS-232 cable to the PC (both downloadable from Canvas). When your design is complete, you will see the ASCII code for characters you type appear on the seven-segment display, and you will also see the characters echo back to *PuTTy*.

4. *Receiver design verification*
Download the VHDL testbench file, SerialRX_tb.vhd, from Canvas and simulate the receiver. Comments in the testbench explain what to expect when you run it.

Make copies of your VHDL sources and the simulation waveform to include with your report.

Show your simulation to one of the lab staff and obtain a signature on your cover sheet.

IN THE LAB

Part II: Modular build and test

Follow these steps in the lab to assemble and test your receiver. To avoid problems, your project hierarchy should only contain the files you are currently using.

1. **Top level file**
   You are provided with a top level VHDL file, `Lab5top.vhd`. Create a new project, add the top level file along with your receiver as a component, and wire it up by port mapping. Synthesize this top level model to make sure that the syntax and connections are correct.

2. **Received signal — oscilloscope.**
   The first step is to make sure you're getting signal into the FPGA from the serial port.

   (a) Plug the Pmod-RS232X into Pmod connector JA. Open a UCF file and assign pin 3 of JA, the `TXD` output from the PmodRS232X, to the receiver input `RsRx`. (Eventually, you will also assign pin 2 of JA, the `RXD` input to the PmodRS232X, to the transmitter output `RsTx`. That is, the Pmod is transmitting to your receiver, and receiving from your transmitter.)

   (b) Temporarily, bypass the receiver module in your top level. Reroute the `RsRx_p` output port from the receiver to the input port RsRx, and in the UCF file, connect `RsRx_p` to the JC connector. In this way, the serial data simply goes into the FPGA and right back out again without passing through any logic. Make a comment in the code so that you can remember how to undo this modification later!

   (c) Connect the Pmod-RS232X to the serial port on the back of the PC with a serial cable. Connect `RsRx_p` to a digital channel of the oscilloscope. Implement your design and configure the FPGA, in the usual way.

   (d) Start the *PuTTy* program on the PC. Configure the connection for 115,200 baud, no parity, one stop bit, and no flow control. Set the oscilloscope for normal trigger, on the falling edge of the `RsRx_p` signal (looking for the start bit). When you tap on the keyboard you should see a waveform like the one at the beginning of this handout.

   The number 1 in ASCII code is 31h = 0011 0001. Type 1 on the keyboard and observe the waveform on the oscilloscope. You should see 0100011001 (start bit, lsb … msb, stop bit), bracketed by a 1 level on either side. As you move up the keyboard from 1 to 9 (ASCII codes 31h through 39h), you should see the waveform change appropriately.

   (e) Follow the instructions in the oscilloscope manual to decode the bit stream and display the characters. Make sure both you and your partner are able to do this.

Make a screenshot of the oscilloscope trace for your report.

Obtain a TA signature on your cover sheet.

3. **Serial receiver: implementation**

   (a) Undo the bypassing (part (b), above) and reconnect the receiver to the top level ports.

(b) Add the multiplexed seven-segment display component to your top level and modify your source and UCF file so that rx_data is displayed on two of the digits.

(c) Implement your design and configure the FPGA. When you type on the keyboard, you should see the ASCII code appear on the seven-segment display.

(d) If you need to debug, you have the signals clk10_p, RsRx_p, rx_shift_p, and rx_done_tick_p . Bring these out to unused pins on Pmod connectors JB through JD, and connect them to the oscilloscope. For example, looking at rx_shift_p and RsRx_p together on the oscilloscope will tell you if you are generating shifts at the right positions along the serial stream.

When you have this working (congratulations!)...

Demonstrate to a TA and obtain a signature on your cover sheet.

4. Serial loopback

By adding a serial transmitter, you can take the data byte from the receiver and loop it back through the transmit port of the RS-232 link to the PC and *PuTTy*, so that what you type on the keyboard will appear on the screen.

(a) Add the transmitter code, SerialTx.vhd, to your project and instantiate it into your top level. Wire the rx_data bus from the receiver to the tx_data input of the transmitter. Bring the tx port of the transmitter to the RsTx port in the top level (and uncomment it). Map the tx_done_tick port to open, which is the proper way to leave something disconnected.

(b) First, we'll do a "manual loopback". Bring the tx_start input to a top level port and label the port tx_start_p. In the UCF file, wire tx_start_p to a pushbutton, and connect pin 2 of JA to the RsTx port. Implement your design and configure the FPGA. Monitor the RXD and TXD lines with the oscilloscope, using the RS-232 decode function.

Now, after you type a character and see the ASCII code on the seven segment display, push the button and you should see many (because the switch isn't a monopulser) instances of the character in the *PuTTy* window and on the oscilloscope screen.

(c) Now "rewire" tx_start to rx_done_tick. This should make the loopback automatic—the receiver triggers the transmitter.

When you are able to type characters on the keyboard and echo them back to the PC (congratulations again!)...

Demonstrate to a TA and obtain a signature on your cover sheet.

AFTER THE LAB

Part III: Postlab

Turn in the following design documentation, stapled together in the order shown.

1. Completed TA Check Sheet, with signatures

2. Design and simulation

(a) All the VHDL files you created, top level and components.

(b) A printout of the simulation waveform for your receiver. Annotate it to help the reader see that it's working the way it should.

(c) The UCF file.

3. Synthesis
Last week you saw how the synthesis report provides information about resource usage on the FPGA. This week we'll review resource usage and also look at timing information.

(a) What high-level units (state machines, registers, counters, etc) were synthesized from your VHDL model?

(b) How many LUTs, how many flip flops, how many slices, and how many I/O buffers are used by your design? How do these numbers compare with the total number of LUTs, flip flops, slices, and I/O buffers are available on the FPGA?

(c) What is the critical timing path, flip-flop to flip-flop, for your design? For this path, what is the delay, and how does it divide between logic and wiring? What is the maximum allowable clock frequency?

4. In-lab testing — attach the oscilloscope printout from the initial serial port test (II.2).

5. Discussion

(a) If something doesn't work by the time you write this report, explain where you think the failure is located.

(b) What problems did you encounter in getting your design to work, and how did you solve them?

(c) Make constructive comments and suggestions for improving this lab.

Lab 5: Serial Communications

Cover sheet

Instructions: Bring this sheet with you to the lab. Have a TA or instructor initial your progress as you complete the in-lab assignments. Attach this sheet to the **front** of your lab report.

Your name: ____________________

Lab partner's name: ____________________Section______

Assignment	TA signature	Date
Receiver passes simulation (I.4)		
Received signal, with oscilloscope (II.2)		
Basic receiver operation (II.3)		
Serial loopback (II.4)		