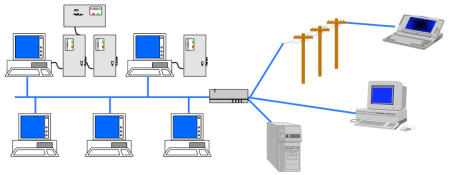
UART Transmitter

In this laboratory exercise, you will learn about the Universal Asynchronous Receiver/Transmitter (or UART) and create a VHDL circuit that implements the transmission portion of this protocol. There is a fair amount of background necessary to implement this lab and this screencast will provide you with the background necessary to design and implement a UART transmitter. Follow the lab manual instructions on when to watch the various parts of this screencast.

Part 1: Overview

In this first part of the screencast I will provide an overview of serial communication, the RS-232 signalling protocol and the UART character frame. The general approach for transmittnig characters using this protocol will be demonstrated.

Serial Communication

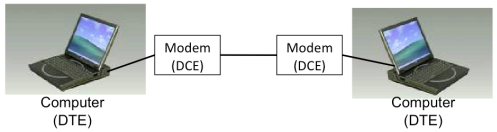


Purpose: motivate the need for communication and provide a historical context

One of the most important components of any computing system is the Input/ Output (or I/O). The I/O infrastructure of a computer allows the computer to communicate with other devices. Input can be received from a device like a disk, a mouse, keyboard, etc. Output can be sent do devices such as a display, the disk, or a network. One of the most important I/O component in a computer system is the component used for communicating with other computers or devices over a network. There are many different methods for communicating between devices. Some methods transmit data in parallel meaning that multiple bits of data are transferred at the same time over a computer bus or a multi-bit cable. Other methods use serial transmission in which only a single bit of data is transferred at a time. In a serial communication system a single physical wire or connection is used to transfer the data. There are advantages and disadvantages to both techniques but most low-bandwidth devices in older computer systems all employed a serial communication protocol.

Future: show pictures of other protocols? (ethernet, GigE, Bluetooth, etc.)

RS-232



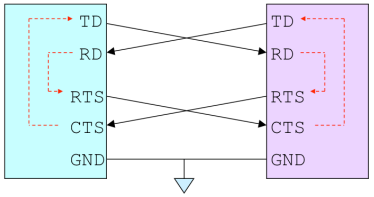
http://en.wikipedia.org/wiki/RS-232

One communication protocol that was widespread within computing systems for many years is the RS-232 protocol which stands for “Recommended Standard – 232”. This protocol was developed for telecommunication systems to transfer data between a Data Terminal Equipment (DTE) and a Data Circuit-Terminating Equipment (DCE). Most early personal computers included one or more RS-232 ports to connect with mice, printers, modems, keyboards, and other I/O devices. The RS-232 grew out of favor in personal computers due to its relatively slow transmission rate, large voltage swing, and large connectors. Most of the functions implemented with RS-232 have been taken over by the Universal Serial Bus or USB. RS-232 is still used today in scientific applications, older equipment, and of course in digital design classes like this.

You can learn more about the history and use of the RS-232 protocol by visiting the wikipedia website on this topic.

Future: pictures of modems and mice

RS-232 Signals

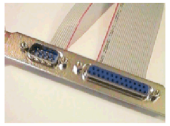


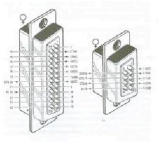
The RS-232 protocol is implemented with a number of signals. Five of them are shown in this figure. These signals are contained within a cable to connect one RS-232 end to another. A common ground is provided between both ends to match the voltage levels of the signal data. The data is transmitted using the TD signal (sometimes called Tx). The data is received using the RD signal (sometimes called Rx). Note that for this point to point RS-232 connection, the TD of one end is connected to the RD end of the other end (i.e., the transmit side of one end is connected to the receve of the other end). This allows both devices on the RS-232 to send and receive simultaneously (this is called full duplex).

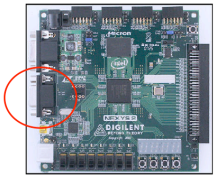
In addition to TD and RD, most RS-232 cable include the signals RTS and CTS. RTS stands for ready to send and it is asserted by a sender indicating that the sender is ready to send data. CTS stands for clear to send and is used by the receiver to indicate that the receiver is ready fro data. These two signals are used for hardware flow control and can be used to stop transmisision when the sender or receiver are not ready. These signals are not supported on the board and we will only use the TD and RD signals.

RS-232 Connectors

DB-25







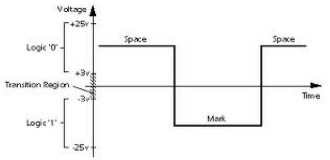
There are two common connectors used for the RS-232 protocol. One is called DB-25 and stands for “D subminature” 25 pin connector. The other is DE-9 and is a 9 pin connector. We will be using the 9-pin DE-9 connector for the lab.

The connector used in the lab is right next to the VGA connector.

Nexys2 Board

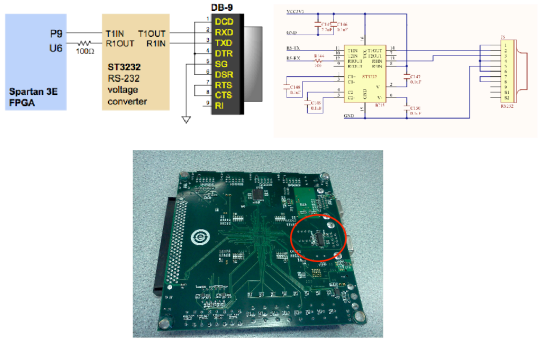
For this lab, the Nexs2 board is connected to your desktop workstation using a DE-9 connector. There is a female connector on your computer and a male connector on your desktop.

RS-232 Voltage Levels



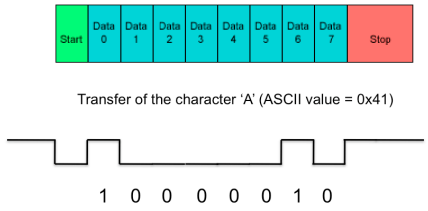
The RS-232 physical protocol does not use the digital logic levels we use within the FPGA. A logic ‘0’ is called a “Space” and is any voltage between +3 and 25V. The logic ‘1’ is called a MARK and is any voltage between -3V and -25 V. Since the RS-232 signalling protocol is not the same as the FPGA digital logic levels, interface logic must be added between the FPGA I/O pins and the DE-9 connector.

RS-232 Voltage Conversion



As shown in this block diagram, a device has been added between the FPGA and the DB-9 connector. The location of this chip is on the back side of the board as shown in this image. This chip, a ST3232, converts the logic from the FPGA into the RS-232 signal. A logic +1 is converted into a -12V and a logic 0 is converted to a +12 V. You do not need to worry about the incompatible voltages of the RS-232 protocol and can simply use a logic 0 and logic 1 when designing your UART code.

Asynchronous Character Framing



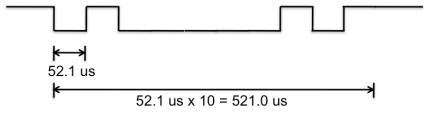
The asunchronous communication will send a single byte or “character” within a frame. There are several components to this frame. The first component is a start bit. The start bit signals the start of a character frame but does not contain any data. The second component of the frame is the data. The data is made up of 7 or 8 bits (we will use 8 bit) and is sent with the least significant bit first. After sending all bits of the character (from least to most significant), is an optional parity bit. We will not use the parity bit but the parity can be used to check the validity of the data. After the optional parity bit is the stop bit. The stop bit signals the end of the character frame. The stop bit also separates the character from the next character in the stream.

Discuss animated example.

Baud Rate

Baud Rate = symbols per second
= 19,200

Baud Period = 1 / Baud Rate
= 1/19,200 = 52.1 us



A key parameter of any asycnronous transfer is the time period for each bit. Both the receiver and the sender must agree to the same bit period to transfer data. For asynchronous transmission, this bit period is specified by the “baud rate” or the number of symbols per second. While any mutually agreeable baud rate can be used, there are a number of acceptable baud rates that are usedn. These include 300, 1200, 2400, 4800, 9600, 19,200, and so on. For this lab, we will use a baud rate of 19,200. The bit period of this baud rate is 52.1 us. Each symbol used in a UART frame is 52.1 us.

In the example from the previous slide, each bit period is 52.1 us and with 10 bit periods (one start, 8 data, and 1 stop) a complete transfer will take 521 us. Note that this is far larger than the clock period of 20 ns. It will take a large number of clock cycles in your FPGA to complete a transfer of a byte.

Receiving a Character

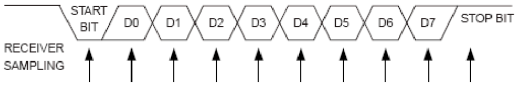


Figure 1. UART Communication Frame Format

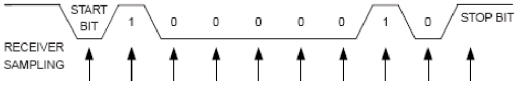
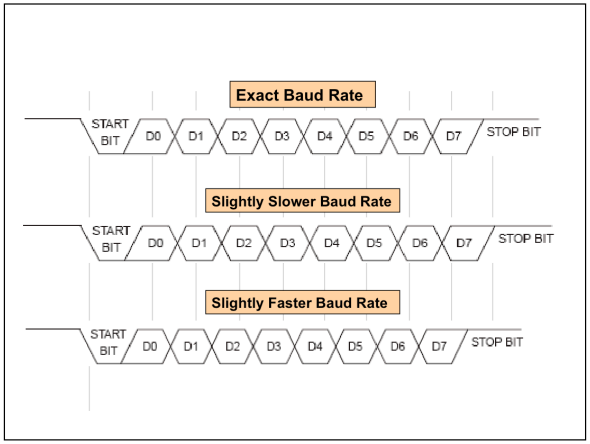


Figure 2. Serial frame of ASCII “A” ($41)

http://www.atmel.com/dyn/resources/prod_documents/DOC0941.PDF
http://www.wcscnet.com/Tutorials/SerialComm/Page1.htm

The receiving end of a UART will be watching the RX signal closely and wait until a start bit is detected. Once the start bit is detected, the receiver needs to extract the data bits from the incoming data stream. The clock on the receiver, however, may not be the same frequency as the clock on the sender (hence the name ‘asynchronous’). This poses no problem since the sender and the receiver have already agreed on a common Baud rate. Once the receiver sees the start of the start bit, the receiver can anticipate the location of each bit in the UART frame.

The receiver will wait until half way through the start bit to start sampling. It will sample the start bit and each suceeding data bit. Although the reeiver and transmitter have agreed on the same baud rate, there may be slight differences in the clocks and timing used by the sender and receiver. The bit timing of the receiver and the sender may be slightly off so the receiver will smaple in the middle of bit period to avoid getting close to the edge. Unless the timing is way off between the sender and the receiver, this sampling in the middle of the bit period will gaurantee that the data is properly sample.d



This figure shows that it is possible for the actual bit period of the transmitter may vary lightly from the specified bit period without any trouble. In this first example, thebaud rate of the sender is the exact baud rate and the reviever samples the bits correctly.

Animate: in the second example, the sender is using a slightly slower baud rate than the receiver is expecting. Notice how the sample period, marked by the lines, is creeping to the left of each bit period. However, all bits are properly sampled because the reveiver is sampling in the middle of the expected baud rate

Animate: in this final example, the sender is using a slightly faster baud rate than the receiver is expecting. In this case the receiver is sampling each bit slightly to the right of the middle. In this case the data is properly transferred.

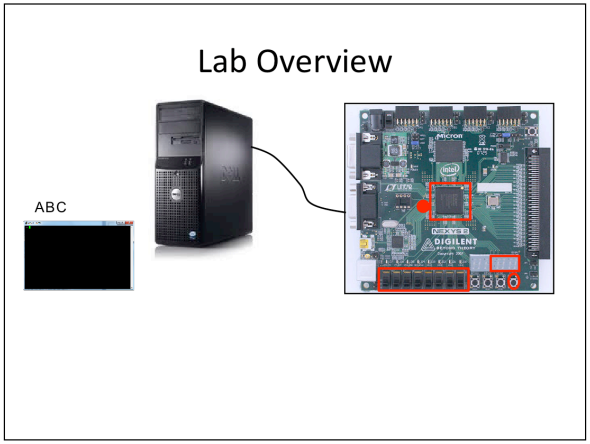
If the baud rate is sufficiently different, the receiver will receive the incorrect value.

At this point we have concluded the overview of the UART communication protocol. You should have a good overview of how data is transferred over RS-232 using UART character frames.

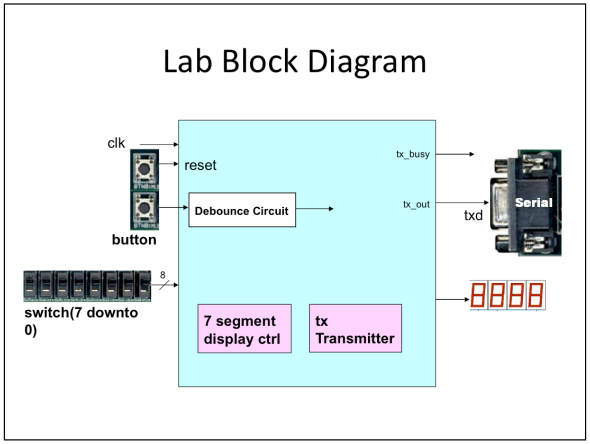
Part 2: Transmitter Design

In this section of the screencast we will provide an overview of the transmitter design. This screencast will provide a lot of detail of the desin and give you sufficient detail to implement a transmitter. The details given here are for your assistance and You are free to implement the transmitter any way you like. There are many ways to implement this transmitter and this screencast is one example.

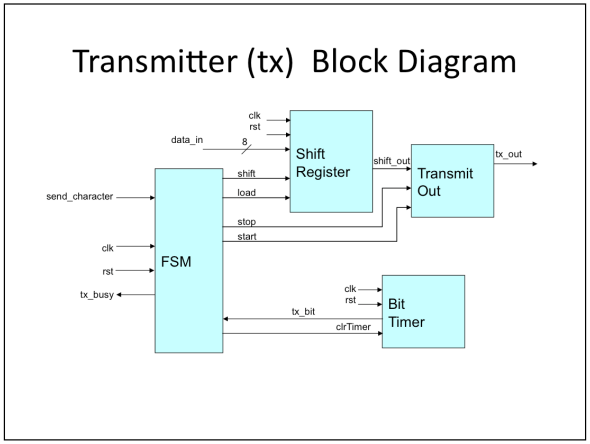
This screencast will provide a fair amount of detail on the transmitter module. You are not required to implement the transmitter as described in this screencast but you are welcome to follow the design as much as you like. This is a fairly complicated design and at this point in the course it may be daunting to complete the design without any guidance. Future labs will not provide this amount of detail.



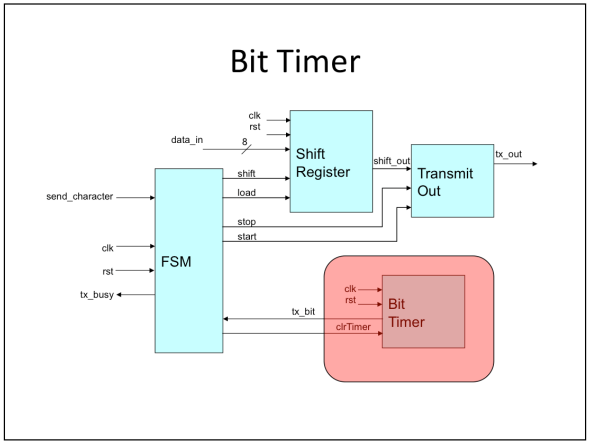
Summarize with an animation how the data is transmitted over the board.



Summarise the architecture of the lab. You will create a new reusable module used for transmitting the character over the uart (this will be the most time consuming part of this lab). You will then create a top-level design that instances your tx transmitter and your seven segment display controller from a previous lab. This top-level design will also have some additional logic for switches, the buttons, and a button debouncer. More details on the debouncer will be given in a later part of this screencast. The rest of this part of the screencast will describe the tx.vhd transmitter module.



The transmitter design can be created with the four blocks: a state machine, a shift register, a bit timer, and a transmit out block. Each of these will be described in more detail. Note that this block diagram does not suggest that you should create four additional entities and instance them. Rather, this block diagram is given to suggest how to partition the design. You should create the logic for all four of these blocks in a single VHDL file using mutiple processes or cother concurrent signal assignment statements. When you hvave completed this design you should have several processes and concurrent statements.



The first block I will discuss is the bit timer. It will generate a signal tx_bit to indicate then end of a bit period and requires a sigmal for clearing the timer.

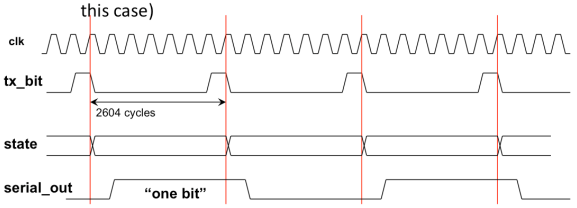
Bit Timer - Transmitter Clocking

- Everything is clocked on same global clock (clk)
- Global clock is 50MHz
- The Bit Timer controls the timing of bits coming out of the serial port.
- Bit Timer needs to create timing pulse at rate of 19,200Hz (52.1 us)
  - That is the baud rate of our serial port
  - Divide factor = 50,000,000/19,200 = 2604.1666…
  - We will use 2604 cycles/baud_period

The purpsoe of the bit timer is to measure bit periods using the synchronous clock. The global clock operates at 50 MHz with a period of 20 ns. This is far smaller than the 52.1 us needed for a single baud period. This bit timer will count long enough to measure one baud period. As shown in this slide, you will need 2604 clock cycles for one baud period (i.e., count from 0 to 2603)

Transmitter Bit Timer

- Bit timer can be created from a counter
  - Should be cleared when clrTimer asserted
    - otherwise increments up to 2603 and rolls over
  - Signal **tx_bit** is asserted during the last cycle (cycle 2603 in this case)



The bit timer should generate a signal, named tx_bit here, that generates a pulse every 2604 clock cycles. This tx_bit signal will be used by your state machine to transition from one serial bit to another.

Your bit timer should support a synchronous clear for resetting the counter. This synchornous clear signal will be used by the state machine to keep your counter at zero when not transmitting. Design your bit timer so that it holds the value zero while the clrTimer is asserted. When clrTimer is not asserted this counter should count from 0 to 2603 and repeat this sequence. On the last count value (2603) your bit timer should assert the tx_bit signal indicating the end of a transmitted bit.

Parameterizable Bit Timer

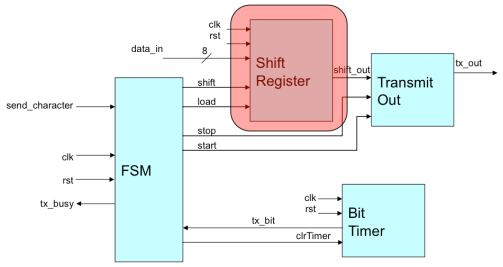
```
generic CLK_RATE : Natural := 50_000_000;
generic BAUD_RATE : Natural := 19_200;
.
.
.
constant BIT_COUNTER_MAX_VAL : Natural := CLK_RATE / BAUD_RATE - 1;
constant BIT_COUNTER_BITS : Natural := log2c(BIT_COUNTER_MAX_VAL);

signal BIT_TIMER : unsigned(BIT_COUNTER_BITS-1 downto 0);
```

- **Use the constant BIT_TIMER_COUNT in the design of your counter to determine when to return to 0**
  - Changes in generics will automatically result in changes to the counter

The 2603 count value is based upon the 19,200 baud rate. To make your circuit more reusable, do not hard code the value 2603 into your VHDL. Instead, use a constant such as BIT_TIMER_COUNT, that indicates number of clock cycles to count for a single bit period. The balue of the bit_timer_count can be dtermined using generics. Provide a generic in your entity for the clock rate and the desired baud rate as shown in this slide. With the clock rate and the baud rate, create a constant that calculates the maximum count value of your bit timer.

You will need to determine the number of bits needed to implement your bit timer. You can compute the number of bits statically by computing the base 2 logarithm of the maximum count value. A function log2c is described in the textbook. Refer to the lab manual for unstrctions on this.

Shift Register



The second module we will discuss is the shift register. The shift register holds the contents of the data that will be shifted out of the transmitter and into the other machine. This shift register should be designed to support a parallel load. When the load signal is asserted, the 8 bit shift register should load the shift register with the 8 bit input data. When the shfit signal is asserted, the shift regsiter should shift right to send another bit of the character.

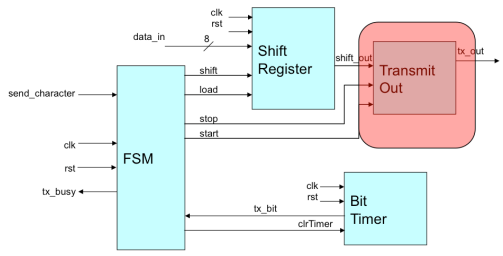
A shift register is used to load the “parallel” data from the switches and “shift” the data out on the serial port

The “LSB” of the shift register drives the “shift_out” signal

load: The shift register should load all data from the switches

shift: The shift register should shift right

Transmit Out



The third piece of this ciruit is th transmit out logic.

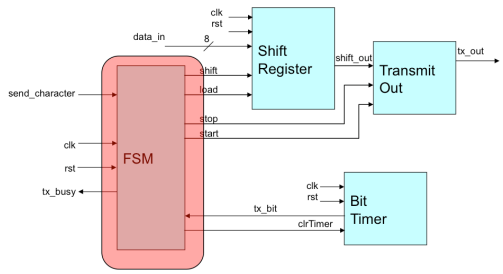
Transmit Out

- Additional logic is needed to make sure the TX signal is valid and clean
  - Outputs a '0' when start is high
  - Outputs a '1' when stop is high
  - Outputs selected data bit from shift_out otherwise
- This signal should be registered in a flip-flop to prevent glitches before leaving the chip
  - The Flip-Flop will remove any transient glitches.
  - The initial value of this flip-flop should be '1' (asynchronous reset to '1' and its initial value in VHDL)

The transmit out logic is used to make sure the TX out signal is valid and clean. This is a very

(Basically read slide)

Transmitter FSM



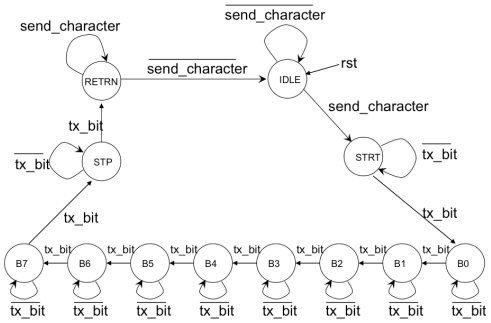
The final block to discuss is the FSM. This is the most complicated part of the module.

Transmitter FSM

- Controls the operation of all elements in the transmitter
  - Controls clearing of Bit timer
  - Controls loading and shifting of shift register
  - Controls transmit out (start bit, stop bit, or shift bit)
- Inputs:
  - send_character: Control signal to initiate transfer
  - tx_bit: Indicates the end of a bit time
- Outputs:
  - tx_busy: Transmitter is busy (middle of transmit)
  - clrTimer: Clear the bit timer
  - load: Load switch data into shift register
  - shift: Shift shift register
  - stop: Send a stop bit
  - start: Send a start bit

Discuss the inputs and outputs. and purpose.

Transmitter State Diagram (Transitions)



Discuss the states in detail (note that the output signals are not annotated)

IDLE:	Waiting for signal to start transmit
STRT:	Issue start bit
B0-B7:	Send a bit (B0 = send bit 0, etc.)
STP:	Send stop bit
RETRN:	Wait for “send_character” to clear

Transmitter FSM Outputs

- tx_busy:
  - true when not in IDLE state
- load:
  - Occurs on transition from IDLE state to STRT state
  - STATE = IDLE and (send_character = ‘1’)
- shift:
  - Occurs on each transition between B_x and B_{x+1}
  - tx_bit = ‘1’ and (STATE = B0 or STATE = B1 . . . or STATE = B7)
- stop:
  - True when in STP, RETRN, and IDLE state
  - Transmitter must drive a ‘1’ when IDLE (i.e. a STOP bit)
- start:
  - True in STRT state
- clrTimer:
  - True in IDLE state (the counter is free running otherwise)

Summarize the outputs.

Summarize the screencast (done)

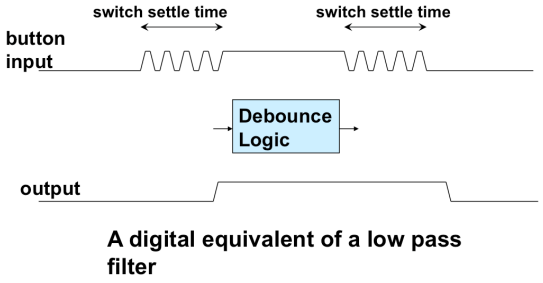
Part 3: Signal Debouncing

Signal Bouncing

- The buttons will "bounce" many times each time the button is pressed
  - Pressing the button generates a very noisy analog signal
  - The corresponding digital signal will transition many times until the button settles
- Additional circuitry must be added to "debounce" the button
  - If this bounce is not removed, one press of the "send character" button will result in many distinct send character signals
  - You will see many characters send to the PC
- You must add a debouncing circuit to this signal

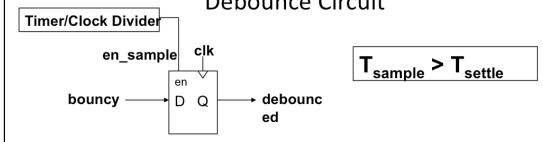
Button Debounce Circuit

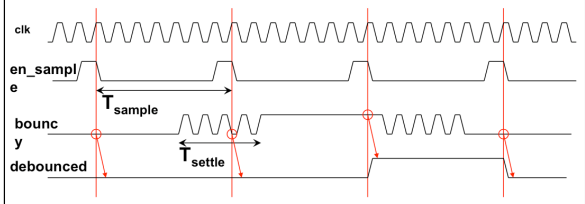
button settle time is typically around 10 msec



A digital equivalent of a low pass filter

Debounce Circuit





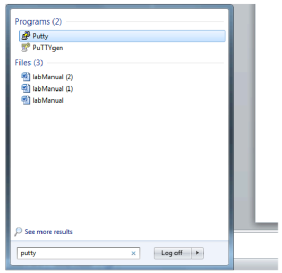
Debounce Circuit

- You can create a debouncer by "sampling" the input signal infrequently (~10 ms?)
  - Create a counter to measure the delay time
  - Sample the input signal every time the counter rolls over
- You should make the count value of the debouncer parameterizable based on the clock rate so the debounce time no matter what the clock rate is set to.

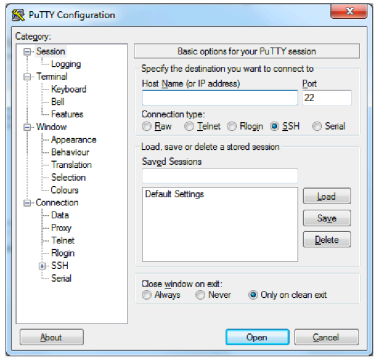
Part 4: Putty Terminal

Once you have designed your transmitter and created a bit file you are ready to try your transmitter on the board. Before doing this you need to setup your worksation and create a terminal that will accept characters sent from your FPGA board.

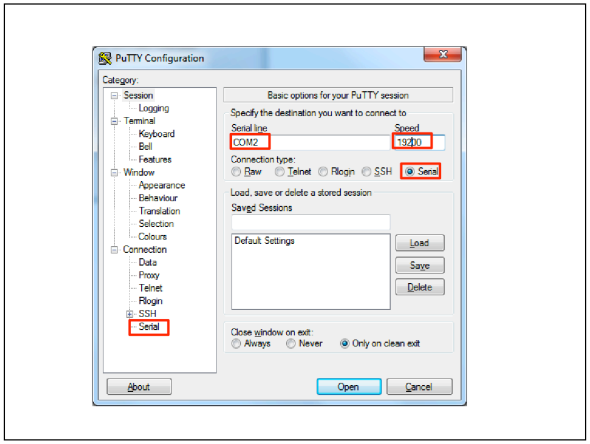
Putty Tutorial



There is a utility on your desktop machine called “putty”. You will create a terminal window within putty to communicate with your FPGA board. Find the “putty” utility on your desktop and open it up.



When you open puttty, the following window appears. This is the putty configuration window and is used to set the configuration options for the terminal that you will use.

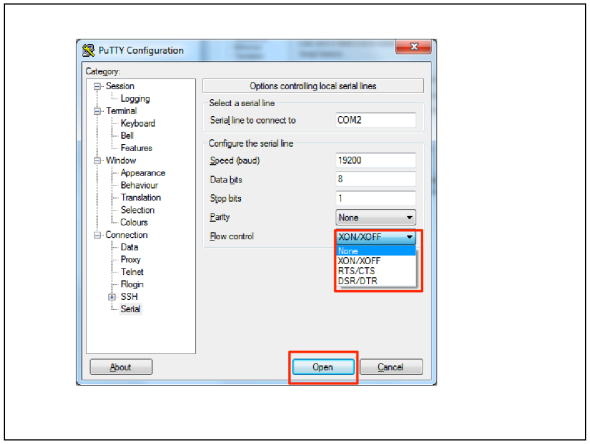


Begin by selecting the connection type. You will be using the “serial” or UART connection type.

The next parameter to select is the serial port to use. Your FPGA boards have been hooked up to the COM2 serial port. Select the COM2 serial port for your terminal.

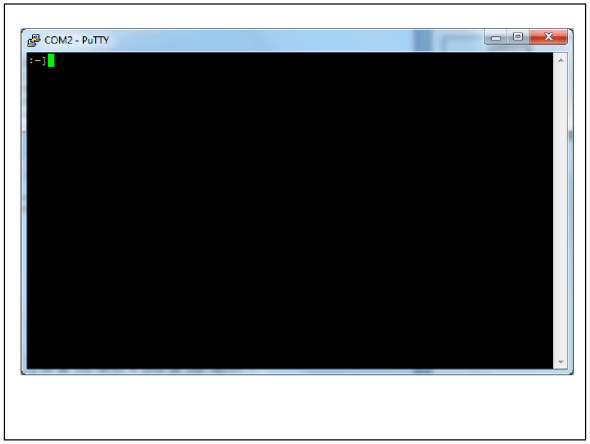
Next, set the baud rate of this connection (19,200)

Finally, select the “Serial” under the connection Category.



When you select “serial” the PuttyConfiguration window will change and provide more details on the serial connection. Make sure that the baud rate is correct (19200) and that the connection is setup for 8 data bits and 1 stop bit. Turn off Flow Control by selecting None.

Finally, create your terminal window by selecting “Open”.



When you click open, a serial terminal window will open. You are now ready to send characters from your FPGA board to the terminal. Send a character from your board to theterminal by specifying the ASCII value of the character you want to send (0x41 for ‘A’) and pressing the send_Character button. A single character should show up on your terminal if your circuit is working properly.

If you have

(check the cables on both ends – FPGA and comjter). Check comm port.