Advanced Logic Synthesis for Electronics
www.ALSE-FR.com

A.L.S.E. Application Note
RS232 & UARTs Basics

Introduction

This physical and logical format has been the most popular way for transmitting data between two computers or between a computer and a peripheral.
It is now less popular on personal computers especially laptops, but very cheap USB-RS232 are widely available if you need to add such one port to any computer lacking one.

This recent obsolescence has many different reasons among which : transmissions today need much higher speed (RS232 was typically limited to ~100k bits/second), external phone-line modems do not exist any more, and USB & wireless are now pervasive for data links and peripherals.

However, this type of link is extremely easy and cheap to implement and only requires one wire in each direction. It is then still a very good candidate for simple connections to / from an FPGA.

The present is now very much focused on Ethernet, and ALSE has developed an excellent technology and many simple to use Tools and IPs, but that's another story.

Principles

RS232 is a character-oriented asynchronous serial protocol. Each character is sent and received serially and independently from the next or previous one, as opposed to frame-based protocols.

This format is asynchronous and self-synchronizing : each character generates its own synchronization information, no other signal (clock) is required, and characters are not synchronized between them.

Data length, presence of parity information, speed (baud rate), and space between characters can be selected to build a format best suited to the application, as we'll see in the next paragraphs.

Data Format

Baud rate

Extremely old systems used 1200 bauds, a speed that practically any equipment was able to reach, so it was long considered as a nice fall back speed, or the lowest speed that was needed to implement.
9600 bauds was a quite popular speed, and **115,200 bauds** was the highest speed supported by most products and became quite popular on PCs (like for the external 33k6 modems).

Data Length

Many values are possible, but the most popular formats are 7 bits (obsolete) or **8 bits**.

Parity

When some data integrity checking is desired, *parity* can be added. However, a much better checking is obtained when the characters are accumulated in a checksum or -even better- in a CRC checker. So the use of parity has never been very popular (50% of chances that a wrong character is undetected).

The “parity” could also be a fixed 0 or 1 value (Mark/Space) for all characters (not very coherent with the name “parity”), and this was even less popular (and useful) than the true odd or even parity !

The Table below does summarize the different possibilities with respect to Parity.

Parity	**Value of last bit transmitted before stop bit**
EVEN	1 if the data transmitted has an even amount of 0 bits.
ODD	1 if the data transmitted has an odd amount of 0 bits.
MARK	1
SPACE	0
NONE	No parity bit is transmitted. Most popular option.

Stop Bit(s)

There is no maximum space between characters, the protocol is **a**synchronous and the line could stay Idle for any length of time. However, the minimum space separating characters is part of the definition of the format. It is typically **1 bit**, but can be also 1.5 or 2 bits. My personal preference goes to 1.5 bits which facilitates the resynchronization in a continuous stream and visual inspection (and is compatible with a 1 stop bit receiver).

Transmission Format Notation

RS232 Transmission format is often abridged under the form : “Speed-Parity-Data bits-Stop bits”.
Example : “**115200-N-8-1**” means 115,200 bauds, No parity, 8 data bits, 1 stop bit.
Other example : “1200E71”.

Physical Interface

RS232 is a **symmetrical voltage** interface (like +/- 15 Volts).

More precisely, positive and negative voltages above and under given (+/- 3V) thresholds are suitable, without having to actually reach 15V. But using 0 .. 5V (eg) is absolutely **not** suitable !
In fact the voltage region comprised between -3V and +3V is considered as “unknown”.

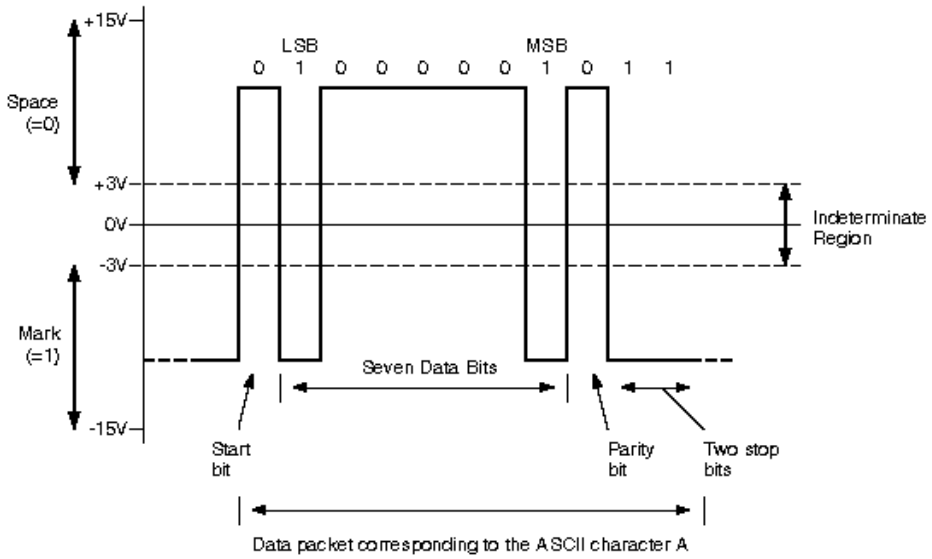

Illustration 1: Example of transmission of character 'A' in 7-E-2 format.

As we can see, the physical format mandates the use of an interface chip (RS232 line driver aka “level shifter”) to transform the usual TTL (or lower) logic signals into positive and negative voltages, and vice versa. These chips now do not even require a negative power supply, only one to four external capacitors since they include a charge pump system to generate the stepped up and negative voltages.

If you look at the RS232 signal on the transmission line, you'll notice that a logic “1” at the logic side corresponds to -10V and a logic “0” to +10V ; there is an “inversion”. For historical reasons, the two levels may be referred to as “mark” for logic 0 and “space” for logic 1.

A consequence of the format is that it is very easy to determine if a signal on an RS232 connector is an input or an output : an Idle output is quite negative, while an input is typically in the indeterminate region.

To transform parallel characters of the expected lengths in a serial stream and back, “UART” chips were typically used (Universal Asynchronous Receive-Transmit). We'll see you can easily design your own.

Important ! For a local (short range) transmission : chip to chip, or FPGA to Micro-Controller etc, you do not have to convert to the high level voltages ! It is perfect just to stay at the **logic** levels.
In fact, level converters are only qualified to maximum bit rates that may not be too high. The best ones are typically qualified at 230k bauds maximum. Without level converters, you can reach higher speeds.

Note that, even if you use RS232 level shifters, you may just ignore the symmetrical voltages and only think of 0's and 1's in the logic level side, as we do in the rest of this document.

Timing Diagram – logic level side

Note :
We are now observing the signals at the UART/FPGA side, not at the +/- 15V physical line (in other words, we look at the FPGA or UART side of the line level converter if one is used).

When transmitting a character :

- a **START** BIT (0) is first sent,
- followed by the **DATA bits** (general 8, but could be 5, 6, 7, or 8 bits), **starting with the LSB** and continuing up to the MSBit,
- followed by a **PARITY** bit (if parity is enabled),
- followed by a **STOP** bit (1) which duration can be 1, 1.5 or 2 bit lengths.

The duration of each bit is equal to (1 sec / Baud_Rate).

The sequence is repeated for each character sent, but the line may remain Idle (=1) for any duration.

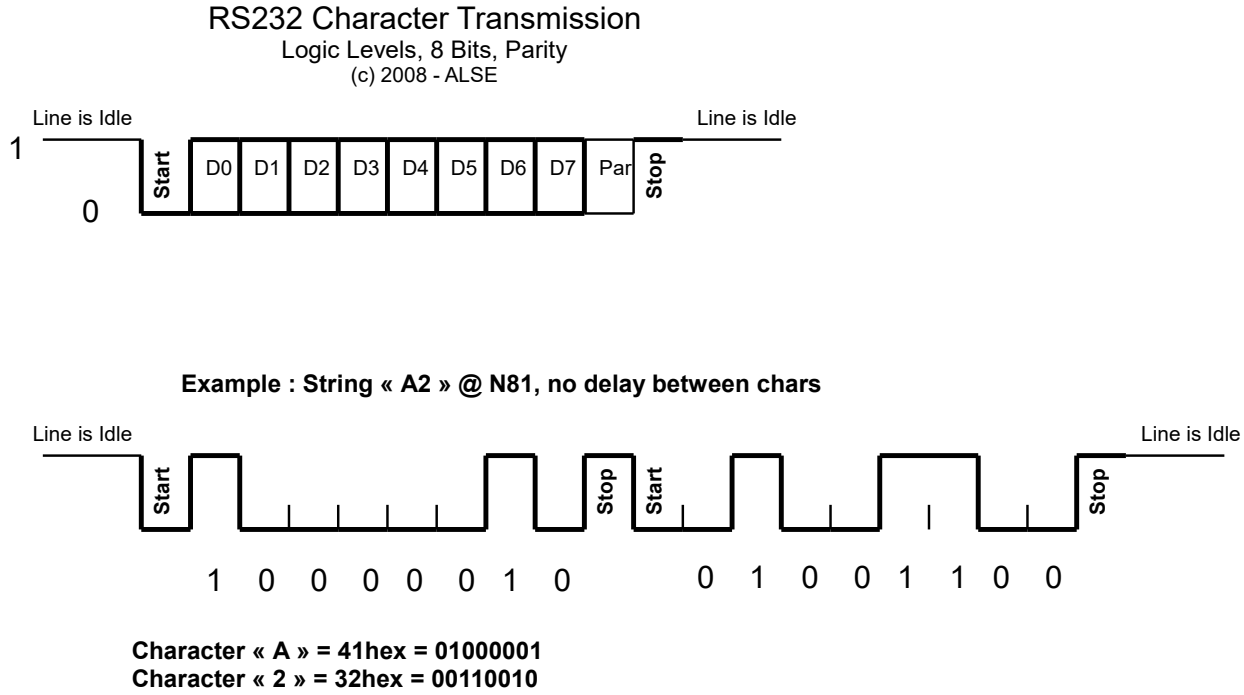

Illustration 2: RS232 Timing Diagram – Logic Levels

Actual transmission speed

From the above diagram, it is easy to deduce that the maximum effective transmission speed, in the case of NO PARITY and EIGHT BITS per character is : **Baud_Rate / 10** (or Baud/10.5 if using 1.5 stop bit).

At 115,200 bauds, the maximum transfer rate is 11,520 bytes per second.

Designing an RS232 Transmitter

This is the “piece of cake” part !

Let take the example of **115200 N 8 1** format.

- Build a timer generating ticks at the (as exact as possible) baud rate (= 8.6805555... us).
- Build a 10 bits shift register with right-most bit connected to the TX output, and initialized to all 1's.

1. To transmit a new character “Byte” : upon a tick, load the shift register with “1-Byte-0”.
2. Shift right ten times at each tick, shifting in a “1” from the left.
3. When done shifting, you're ready to send another character.

Slick. Designing the transmitter is typically a 15 minutes task in HDL, including verification.
Even with the few steps involved, it's still a very good idea to implement the Transmitter using a Finite State Machine (and a counter variable to count down from 10 to 0).

Designing an RS232 Receiver

This is the (slightly) more difficult part. Let take again the example of **115200 N 8 1** format.

You must have a tick generator that can be restarted on demand and generating ticks at half the bit period. And a Shift Register.

1. Keep your Tick Generator at zero, waiting until the line gets low (beginning of Start bit).
2. Loop 10 times :
   (a) Wait for tick (mid-bit), Right-Shift in the Data Line (Rx) in your register.
   (b) if reached last (stop) bit exit the loop, else just Wait Tick (end of bit).
3. End of Loop : the register's MSB must be “1” (stop bit), else you have a problem (line may not be synchronized, bad baud rate or format etc...).
   The character is in the middle of your shift register, you can pass it to the next block (in charge of processing the characters) or push it in a FIFO, etc...
   Return to state 1.

Obviously, this is very simple to implement as a Finite State Machine.

Note : legacy UART *chips* (like the famous 8250 followed by the 16550) used another method : over-sampling (by x 16 min), to detect the edges and sample the bits. This was supposed to add some tolerance to glitches or noisy lines, but it wasn’t very efficient and it had the very painful consequence of having to use special frequency Crystals to handle high baudrates. If you find a 1M832 or a **14.7456 MHz** Crystal in an Atmega board schematics, this is the reason !

Going further

ALSE's website contains more information about how to implement a UART, the elements of a behavioral model, a sample application etc... Our Tornado FPGA board has two RS232 ports with appropriate level shifters and can be used to quickly implement and test a UART.

I hope you enjoyed this quick summary and you are now convinced that RS232 is still a very simple and very efficient method to exchange data with an FPGA at speeds up to several Megabits/s.

Bertrand Cuzeau – CTO A.L.S.E - info at alse-fr dot com