Timing Errors in Serial Communication

On the previous page, we saw digital logic traces of asynchronous serial communication. The receiver must read the input pin at a specific rate to properly interpret the incoming bit patterns. In asynchronous communication, the sender and receiver rely on their own timing circuitry, rather than sharing a clock. On this page, we'll determine how inaccurate the timing can be such that the devices can still communicate reliably.

Baud vs. BPS

In the case of two devices using a simple serial digital signal, the data rate can be described as either baud or bits per second (bps or bit/s). Think of baud rate as the switching speed, and bps as the switching speed times the amount of information present at each switch. If only a single bit of information is available at each switch, then baud and bps are the same. However, if the signal is converted to parallel or an analog value, then the baud rate and bps will be different.

Imagine you have a bass drum. At a steady rate, you either hit the drum or stay quiet. Hitting the drum is a '1' bit; staying quiet is a '0' bit. That's our simple serial digital signal where the baud (beats per second) is identical to the bps (amount of information delivered per second).

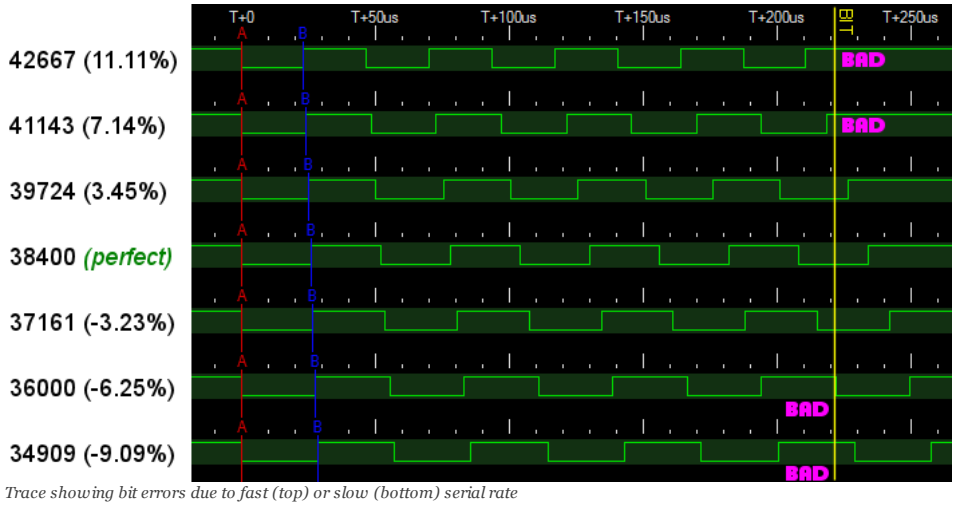
Now, imagine you have a piano with eight keys. If you play the piano at a steady rate, you can deliver 8 bits per beat by pressing up to 8 keys at the same time. In that case, the bps rate (amount of information delivered per second) is eight times more than the baud rate (beats per second).

Besides analog conversion and parallel wires, if you add compression to the communication channel then the bps can go much much higher than the baud rate. You are increasing the amount of information delivered per second even if the timing of the electrical switching stays the same.

How Far Off is Acceptable?

The receiver is expecting to receive serial data at a specific rate. When the wire voltage initially changes from high to low (the start bit), the receiver starts a timer. Rather than reading each bit at the beginning of its time, a good receiver will read the value about halfway through. In doing so, any jitter or slight timing mistakes in changing states at the beginning or ending of a bit won't corrupt the value. (In fact, a really good receiver may take multiple samples and average the value, to eliminate electrical noise or spikes.)

Let's see what happens when the receiver has perfect timing but the transmitter is either too fast or too slow. We're going to focus on the final data bit, shown by the yellow line. The data bit should be '0' -- represented by a low voltage. On the 38400 (perfect) row, the yellow line passes through the center of a low signal.



Trace showing bit errors due to fast (top) or slow (bottom) serial rate

On the top row (42667 bps), the transmitter is so fast that it is already sending the stop bit (high) at the moment in time that the receiver is expecting to read the final data bit (low). On the bottom row (34909 bps), the transmitter is so slow that it is still in the process of sending the next to the last data bit (high) at the moment in time that the receiver is expecting to read the final data bit (low). In fact, the final data bit is read as '1' on the top two rows and the bottom two rows.

To make it easy to see bit drift, the above image illustrates transmission speed errors when sending the letter 'U' (each subsequent bit goes from high to low). That might have you thinking a good receiver could reset its clock at the start of every voltage change to constantly resynchronize the timing. However, not only would electrical noise (spikes and dips) result in increased timing errors, but, as you'll recall from the previous page, some data patterns have long runs of all zeros or all ones. Put simply, you'll introduce other errors for sake of eliminating some errors. It is better to have an accurate clock.

Calculating Satisfactory Serial Timing

For 10 bits (1 start, 8 data, 1 stop), each bit is 10% of the total time. Therefore, if the difference between the timer and receiver is more than 10%, then an entire bit has occurred too early or passed by without being read, regardless of the receiver's sampling method. Clearly a 10% difference in timing is unacceptable.

For purposes of this analysis, let's say the receiver reads the bit value at the halfway point. So, as long as less than half a bit is too early or late, the data will still be communicated correctly. Half a bit is 5% of the total time.

If the sender is 5% too fast, and the receiver is 5% too slow, we're back to 10% difference again and a whole bit is missing. Therefore, to be fair, each device is only allowed to be less than 2.5% in error, to ensure that the total difference in timing between devices is less than 5%. To provide a safer margin, Atmel recommends a 2% error or less per device.

Atmel AVR UBRR Serial Values

To help you out, here is a massive table with crystal values in MHz, and the bps (or baud) rates that can be achieved. If you have an Atmel AVR microcontroller, you simply need to set UBRR to the integer value on the left side of the bps column. For example, to achieve 38400 bps with 0% error and an 18.432 MHz clock, UBRR should be set to 29.

MHz↓ bps→	300		600		1200		2400		4800		9600		14400		19200		28800		38400		57600		76800		1
0.128000	26	-1.2%	12	2.6%	6	-4.8%	2	11.1%	1	-16.7%	0	-16.7%	0	-44.4%	-1	too fast	-1	too fast	-1	too fast	-1	too fast	-1	too fast	-1	to
1.000000	207	0.2%	103	0.2%	51	0.2%	25	0.2%	12	0.2%	6	-7.0%	3	8.5%	2	8.5%	1	8.5%	1	-18.6%	0	8.5%	0	-18.6%	0	-4
1.843200	383	0.0%	191	0.0%	95	0.0%	47	0.0%	23	0.0%	11	0.0%	7	0.0%	5	0.0%	3	0.0%	2	0.0%	1	0.0%	1	-25.0%	0	
3.579545	745	0.0%	372	0.0%	185	0.2%	92	0.2%	46	-0.8%	22	1.3%	15	-2.9%	11	-2.9%	7	-2.9%	5	-2.9%	3	-2.9%	2	-2.9%	1	
3.600000	749	0.0%	374	0.0%	187	-0.3%	93	-0.3%	46	-0.3%	22	1.9%	15	-2.3%	11	-2.3%	7	-2.3%	5	-2.3%	3	-2.3%	2	-2.3%	1	
3.686400	767	0.0%	383	0.0%	191	0.0%	95	0.0%	47	0.0%	23	0.0%	15	0.0%	11	0.0%	7	0.0%	5	0.0%	3	0.0%	2	0.0%	1	
4.032000	839	0.0%	419	0.0%	209	0.0%	104	0.0%	52	-0.9%	25	1.0%	17	-2.8%	12	1.0%	8	-2.8%	6	-6.3%	3	9.4%	2	9.4%	1	
4.096000	852	0.0%	426	-0.1%	212	0.2%	106	-0.3%	52	0.6%	26	-1.2%	17	-1.2%	12	2.6%	8	-1.2%	6	-4.8%	3	11.1%	2	11.1%	1	1
4.194304	873	0.0%	436	0.0%	217	0.2%	108	0.2%	54	-0.7%	26	1.1%	17	1.1%	13	-2.5%	8	1.1%	6	-2.5%	4	-9.0%	2	13.8%	1	1
4.433619	923	0.0%	461	0.0%	230	0.0%	114	0.4%	57	-0.5%	28	-0.5%	18	1.3%	13	3.1%	9	-3.8%	6	3.1%	4	-3.8%	3	-9.8%	1	2
4.915200	1023	0.0%	511	0.0%	255	0.0%	127	0.0%	63	0.0%	31	0.0%	20	1.6%	15	0.0%	10	-3.0%	7	0.0%	4	6.7%	3	0.0%	2	-1
7.372800	1535	0.0%	767	0.0%	383	0.0%	191	0.0%	95	0.0%	47	0.0%	31	0.0%	23	0.0%	15	0.0%	11	0.0%	7	0.0%	5	0.0%	3	
7.680000	1599	0.0%	799	0.0%	399	0.0%	199	0.0%	99	0.0%	49	0.0%	32	1.0%	24	0.0%	16	-2.0%	12	-3.8%	7	4.2%	5	4.2%	3	
8.000000	1666	0.0%	832	0.0%	416	-0.1%	207	0.2%	103	0.2%	51	0.2%	34	-0.8%	25	0.2%	16	2.1%	12	0.2%	8	-3.5%	6	-7.0%	3	
9.216000	1919	0.0%	959	0.0%	479	0.0%	239	0.0%	119	0.0%	59	0.0%	39	0.0%	29	0.0%	19	0.0%	14	0.0%	9	0.0%	7	-6.3%	4	
10.000000	2082	0.0%	1041	0.0%	520	0.0%	259	0.2%	129	0.2%	64	0.2%	42	0.9%	32	-1.4%	21	-1.4%	15	1.7%	10	-1.4%	7	1.7%	4	
11.059200	2303	0.0%	1151	0.0%	575	0.0%	287	0.0%	143	0.0%	71	0.0%	47	0.0%	35	0.0%	23	0.0%	17	0.0%	11	0.0%	8	0.0%	5	
13.516800	2815	0.0%	1407	0.0%	703	0.0%	351	0.0%	175	0.0%	87	0.0%	58	-0.6%	43	0.0%	28	1.1%	21	0.0%	14	-2.2%	10	0.0%	6	
14.745600	3071	0.0%	1535	0.0%	767	0.0%	383	0.0%	191	0.0%	95	0.0%	63	0.0%	47	0.0%	31	0.0%	23	0.0%	15	0.0%	11	0.0%	7	
15.360000	3199	0.0%	1599	0.0%	799	0.0%	399	0.0%	199	0.0%	99	0.0%	66	-0.5%	49	0.0%	32	1.0%	24	0.0%	16	-2.0%	12	-3.8%	7	
16.000000	3332	0.0%	1666	0.0%	832	0.0%	416	-0.1%	207	0.2%	103	0.2%	68	0.6%	51	0.2%	34	-0.8%	25	0.2%	16	2.1%	12	0.2%	8	
16.384000	3412	0.0%	1706	0.0%	852	0.0%	426	-0.1%	212	0.2%	106	-0.3%	70	0.2%	52	0.6%	35	-1.2%	26	-1.2%	17	-1.2%	12	2.6%	8	
18.432000	3839	0.0%	1919	0.0%	959	0.0%	479	0.0%	239	0.0%	119	0.0%	79	0.0%	59	0.0%	39	0.0%	29	0.0%	19	0.0%	14	0.0%	9	
19.660800	4095	0.0%	2047	0.0%	1023	0.0%	511	0.0%	255	0.0%	127	0.0%	84	0.4%	63	0.0%	42	-0.8%	31	0.0%	20	1.6%	15	0.0%	10	
20.000000	4166	0.0%	2082	0.0%	1041	0.0%	520	0.0%	259	0.2%	129	0.2%	86	-0.2%	64	0.2%	42	0.9%	32	-1.4%	21	-1.4%	15	1.7%	10	
22.118400	4607	0.0%	2303	0.0%	1151	0.0%	575	0.0%	287	0.0%	143	0.0%	95	0.0%	71	0.0%	47	0.0%	35	0.0%	23	0.0%	17	0.0%	11	
AVR↑ error↑																										

Did you notice the rows with the most accurate bps values have unusual crystal MHz? What's magical about 1,843,200; 3,686,400; 7,372,800; 9,216,000; 11,059,200; 14,745,600; 18,432,000; and 22,118,400? Those crystals and the bps rates are all evenly divisible by 300.

Put another way, 1,843,200 MHz ÷ 300 bps = 6144. The transmitting microcontroller hardware running at 1,843,200 simply counts down from 6144, outputs a bit, and repeats. But, if the microcontroller was running at 1,234,567 MHz, it would need to count down from 4115.223333333333, which is not possible with integer hardware.

The other thing magical about those crystal values is that they are readily available. There's no use calculating the baud rate possible for a crystal frequency that you can't buy.

If you are building two devices from scratch that talk to each other, there is no reason that you need to limit yourself to classic baud rates in multiples of 300. There is nothing special about those rates. As long as the devices don't need to talk to a computer or some other off-the-shelf device, you can pick any crystal value and any serial timing. For example, a device talking at 123456 bps to a receiver at 123456 bps has 0% relative frequency error.

Is there a way for the receiver to automatically detect the serial data rate in order to adjust its timing to the transmitter? Yes! Let's find out how...

« Previous Page | **Next Page »**

books contact legal home