Using Xbee at 115.200 Baud

Several posters to this forum have mentioned that serial communication between an XBee and its host can be unreliable at the 115,200 baud setting. The solution ("use two stop bits") is known and has been posted. However, this update adds a second solution.  
  
While musing on this point, I decided to sit down and do some sums, to see exactly why the unreliability occurred and in what situations the two stop bits would be needed. I wrote up the results for my own notes, and it also occurred to me that it might be worth posting the results here for the archive. Hence this rather long post. I hope someone might find it helpful.  
  
My thanks go to the kind folks at Digi, who responded very promptly and helpfully when I asked for confirmation of details of the XBee operation. Any errors in this post are mine alone: comments and especially corrections are very welcome. I will be very surprised if it turns out that I've got all this right on the first cut. (Later note: I didn't get it right. Hence this update.)  
  
\*Changes in this Update\*  
  
There are two changes since the original post in May 2009.  
1. In the original post I had completely overlooked one blindingly obvious solution to the problem. I've now rectified that by adding the section "Solution Using Host Baud Rate".  
2. In the original post while musing on the use of stop bits to slow down the faster transmitter I had claimed that the ATNB=3 setting (mark parity) would have the effect of setting the XBee to two stop bits. I was wrong, and I've now corrected that. Even if possible it would only have been useful at certain non-standard baud rates and nobody complained, so I hope the correction arrives before the error causes anyone too much grief.  
  
\*Baud Rates and their Accuracy\*  
  
In serial communications, the baud rate should be about the simplest parameter you can get. But it isn't. The reason is that what you set is not always what you get. You may think you've specified a particular rate, but the actual rate may be different to a degree that affects the quality of communication.  
  
\*System Clock Frequency and UART Clock\*  
  
In pretty much any digital device, there will be one oscillator (clock) that controls the timing of events. The clock is a signal source, running at a particular high frequency. Any circuitry that needs to use a lower frequency will use a sub-multiple of that frequency. The sub-multiple will be provided by a divider circuit, so any frequency which is an exact integer division of the master frequency is available.  
  
From that, it follows that if you want a frequency that isn't a sub-multiple of the master clock frequency, you're going to have to settle for the nearest approximation. And it's here that the world tends to split into two camps. In one camp we have the PC serial port, which will normally use a crystal clock that does divide exactly to give the advertised baud rates. In the other camp we have microcontrollers, which often use clocks that are a multiple of 1MHz. These clocks do not divide down to give exact baud rates. The XBee itself is in the 1MHz-multiple camp: it has a 16MHz crystal clock.  
  
The UART itself needs a clock signal with a frequency of 16 times the baud rate (see Receiver Operation below). It's this frequency that must be derived from the system clock by the divider circuit.  
  
\*Available Baud Rates\*  
  
As an example, take a system with an 8MHz master clock. If this clock is to drive a UART at 9600 baud, it must be divided down to a frequency of 16 times 9600 = 153,600 Hz. The dividing factor is therefore 8,000,000 / 153,600 = 52.0833 . That isn't an integer value, but the nearest integer is 52. The actual baud rate is then 8,000,000 / 16 / 52 = 9615.38 baud, which is a ratio of 1.0016 (0.16%) from the exact value. That's pretty good, and reliable communication can be expected.  
  
Now suppose we want 115,200 baud from the same master clock. The UART clock frequency must be 16 times 115,200 = 1,843,200 Hz. So the dividing factor is 8,000,000 / 1,843,200 = 4.3402 . The nearest integer is 4, so the actual baud rate will be 8,000,000 / 16 / 4 = 125,000 baud. That differs from what was wanted by a factor of 1.085 (8.5%), which is not at all good. This UART will not be able to communicate with one that runs accurately at 115,200 baud.  
  
To fix this, suppose we double the master clock speed to 16MHz. The dividing factor is now 16,000,000 / 1,843,200 = 8.68, or 9 when we take the nearest integer value. The actual baud rate is therefore 16,000,000 / 16 / 9 = 111,111 baud. The difference ratio is 1.0368 (3.68%), which is in theory just within acceptable tolerance (see below). The master clock of the XBee is 16MHz, so this scenario represents the situation where an XBee is connected to a PC.  
  
Overall, lower baud rates mean higher factors by which the master clock is divided. In turn, that means there is finer control over the frequency. So if we're going to see problems, they're going to happen at the highest baud rates.  
  
\*Serial Line Receiver Operation\*  
  
When a serial line is idle, it is in mark state (logic 1). Transmission of a byte begins with the start bit, which is a period of logic 0. Then come the 8 data bits, and finally the stop bit which is at logic 1 level (same as the following idle state).  
  
The receiver must recognise the beginning of the start bit, and then sample the line halfway through each of the 10 bit periods. The receiver clock runs at 16 times the baud rate, and the receiver can sample the input line once per period of this clock. The sequence of operations is:  
  
1. The start bit begins.  
2. At its next clock cycle, the receiver detects that the start bit has begun. This may be up to 1/16 of a bit period after the actual start.  
3. After another 8 cycles, the receiver samples the line again. If the line is still at logic 0, the start bit is confirmed. Otherwise the initial transition is dismissed as noise.  
4. After another 16 cycles the receiver samples the line. This is repeated a further 7 times, to get the values of the eight data bits.  
5. After another 16 cycles the receiver samples the line again, expecting to see the logic 1 level of the stop bit. If it doesn't see a logic 1 at that point, it discards the data and reports a framing error.  
6. After one more cycle the receiver starts sampling the line at every cycle, waiting for the next start bit.  
  
From that, it follows that a receiver can accept transmitted bytes if they arrive no faster than at periods of 1 + 8 + (8 \* 16) + 16 + 1 = 154 of its clock cycles. If the transmitter and receiver clocks are at the same frequency, the receiver will see a byte every 160 clock cycles. Therefore the transmitter can be faster than the receiver by a factor up to 160 / 154 = 1.039 (3.9%), and in theory all will still be well.  
  
It also follows that if the transmitter is slower than the receiver, all will be well if the period between the beginning of the start bit and the beginning of the stop bit (144 clock cycles for the transmitter) corresponds to more than 153 of the receiver's clock cycles. So the transmitter baud rate must be greater than 144 / 153 = 0.94 of the receiver baud rate.  
  
Any bidirectional serial channel, if there is a baud rate mismatch, will have the transmitter running faster in one direction and the receiver running faster in the other. The tighter condition for the speed difference is for the direction in which the transmitter is running faster, so the ratio of 1.039 is the limiting case for the connection as a whole.  
  
\*Signal Quality\*  
  
We've seen that connecting an XBee to a PC at a nominal rate of 115,200 baud with one stop bit results in a connection which is close to the theoretical limit. In theory it should still work perfectly well, but in practice there are reports of problems with this setup. It seems to me that the likely reason is signal degradation, probably arising from the connecting cable. This cable will have capacitive and inductive qualities, so the signal will be slightly distorted when it emerges at the other end. And since the connection is operating so close to the limit, any distortion or other noise is likely to be fatal to the communication quality.  
  
\*Solution Using Host Baud Rate\*  
  
This form of solution can be applied when the host has a nice fast master clock available. In such a case, and a PC is an example, you can probably set the host to match the actual speed of the XBee: so for nominal 115,200 baud communication you would configure the host to send at the XBee's actual speed of 111,111 baud. I've tested this on a PC and it works for me.  
  
\*Solution Using Stop Bits\*  
  
Theory says that if you connect an XBee to a PC at 115,200 baud you should be able to get away with one stop bit. In practice, any element of noise or other signal degradation is likely to lead to data loss. The baud rates are already set to the best obtainable values, so to prevent the problem the only remaining option is to configure the PC to send two stop bits, thus slowing its transmission rate to one that the XBee can reliably accept. This solution is reported to work in practice and it does work when I try it.  
  
Since at 115,200 baud nominal the XBee is actually running more slowly than the PC, there is no need to slow it further. So there is no need for the XBee to be configured with two stop bits. That's just as well, because the XBee cannot be configured with two stop bits. In the first version of this post, I wrote that the ATNB=3 command would have the same effect. It does when the XBee is transmitting, but the XBee then rejects bytes sent with just one stop bit so the suggestion does not work in practice.  
  
If instead of a PC the XBee is connected to a microcontroller, and if the microcontroller's clock is a multiple of 1MHz, then the whole problem goes away. It doesn't matter that neither end can support standard baud rates exactly, because they can still both be set to an identical non-standard rate.  
  
For microcontrollers with clocks running at frequencies other than those discussed, it's probably a good idea to use the calculations given above to see what baud rate mismatch if any can be expected. If a mismatch is found which is a significant fraction of the limit, then action along the above lines is likely to be needed.  
  
\*Historical Note\*  
  
If you google for this sort of stuff, you'll find statements to the effect that serial line communications can tolerate as much as 10% in the mismatch of baud rates - a figure which appears to contradict the 3.9% figure derived above. The 10% figure in fact goes back to the days of mechanical teletypes, which used the stop bit purely as a delay mechanism between transmitted characters. Teletypes did not check the validity of the stop bit, but modern UARTs do. Teletypes, by their nature, also took an average of the line state during a bit period. And they didn't have the granularity issue of the 16x clock. These differences, and probably some others that I haven't thought of, contribute to the slightly surprising conclusion that a modern UART does in fact have a tighter speed match requirement than the old equipment did.