Communications - The Nitty/Gritty

It is all very well to discuss communications protocols but it is also necessary to examine the real way in which data is passed along a data link in terms of the voltages and signal patterns. We already had a very brief look at the Ethernet hardware earlier, now we look a bit more thoroughly at some simple systems.

In this section we shall focus upon serial links which pass multiple bits of information in a time-sensitive manner. The major differences in these links are in the manner in which the timing is considered and there are essentially three different types:

- asynchronous
- synchronous - separate clock
- synchronous - embedded clock

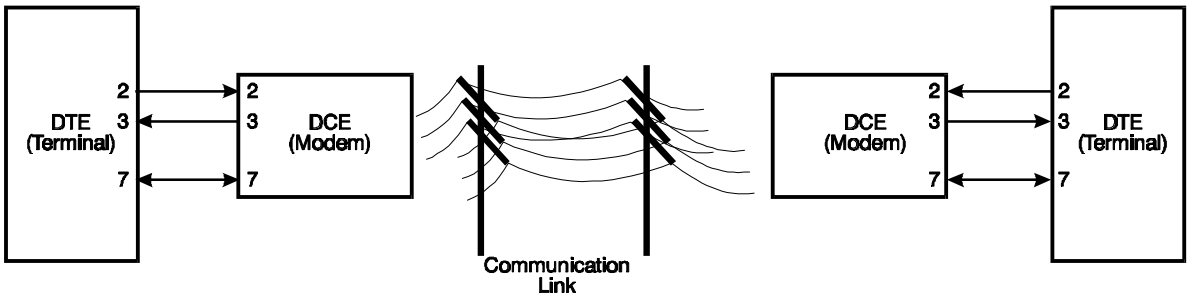
Before discussing these however it is also necessary to consider the voltage levels and other signals which are present on the link. The major serial link system in use today is the so-called "RS-232" standard. (Which is about as non-standard as anything that I know.) The "true" standard refers to a voltage and mechanical interface and is properly the "RS-232-C" standard. It is roughly equivalent to the "CCITT V.24" and "CCITT V.28" standards. These standards define a set of mechanical specifications (will the plugs fit together and are the pin assignments the same), a set of signal definitions (if I call this signal Clear To Send (CTS) what should it do) and a set of electrical specifications (what voltage levels and impedance levels should I drive this line at). However the field is rather more open than that because many manufacturers started making equipment before the standards were introduced and some have their own interpretations of the signals. Thus we get statements like the following:

> In half-duplex modems, even when used on full-duplex facilities, Clear to Send is merely a delayed version of Request to Send. The modem interface control asserts Request to Send and a timer in the modem, upon detecting this assertion, waits for up to 200 milliseconds (depending upon option arrangements) and then asserts Clear to Send back to the modem interface. In this case, Clear to Send is really "Probably Clear to Send". (J.E. McNamara "Technical Aspects of Data Communication, Digital Equipment Corporation).

The standards define the communication between Data terminal Equipment (DTE) (a.k.a. "Customer Provided Equipment" (CPE) or "terminal") and Data Communication Equipment (DCE) (a.k.a "data set" or "modem"). The three most important wires being:

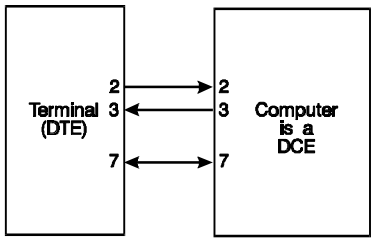
7	Signal Ground
2	Transmitted data FROM DTE TO DCE
3	Received data FROM DCE TO DTE

The designers of this standard had in mind the pattern

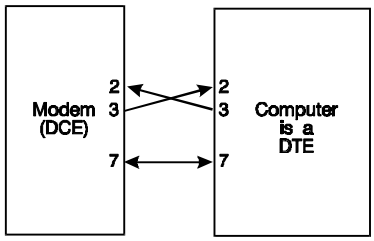


The "True RS232" Hook-Up

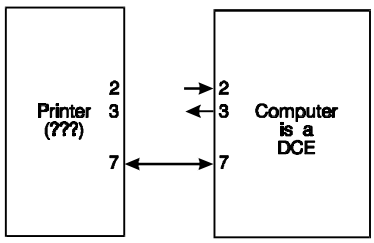
where the wires should be "1-to-1" ie. pin2 to pin2 etc.



Those of you who have used this "standard" will already be familiar with the problem "Is the computer a DTE or a DCE?" If you connect a terminal straight to it, it should be a DCE, but if you attach a modem to it for remote communications it should be a DTE!!!! and if you get that sorted out - what's a printer?



Having realised the problem and understood it, it is generally possible to get these things sorted out with the aid of a service manual.



However this is not the end of the problem as there are many other lines defined in the standards which may or may not be used or ignored. The best advice that I can offer is to very carefully read the instructions for any device you may use and be prepared for trouble. Thus many modems require the use of the Data Terminal Ready signal (DTR) signal even

though the terminal generally asserts it when the power is turned on and does nothing with it otherwise.

Which brings us to the question of the voltage and impedance levels of the system. The standard defines two levels "MARK" and "SPACE". Briefly the requirements are:

Outputs:

- open-circuit voltage in the range ± 25V
- output voltage 5→15V for SPACE, -5→-15V for MARK for a resistive load in the range 3000-7000Ω.
- shall survive a short-circuit between any two pins (including two outputs) and shall pass a current of < 0.5A under those conditions
- shall have a rate of change of output of less than 30V/μS

Inputs:

- shall have an impedance in the range 3000-7000Ω
- shall respond to voltages > 3V as a SPACE and < -3V as a MARK.
- shall survive input voltages in range ± 25V

These standards are reasonably attainable and are followed by most people particularly since there are RS-232 driver and receiver chips available which meet these specs. In fact not using such chips is reprehensible since they guarantee meeting the specifications.

Notice that the MARK state of the line is the lower state in voltage - this leads to confusion as it is counter-intuitive when using an oscilloscope to monitor a line.

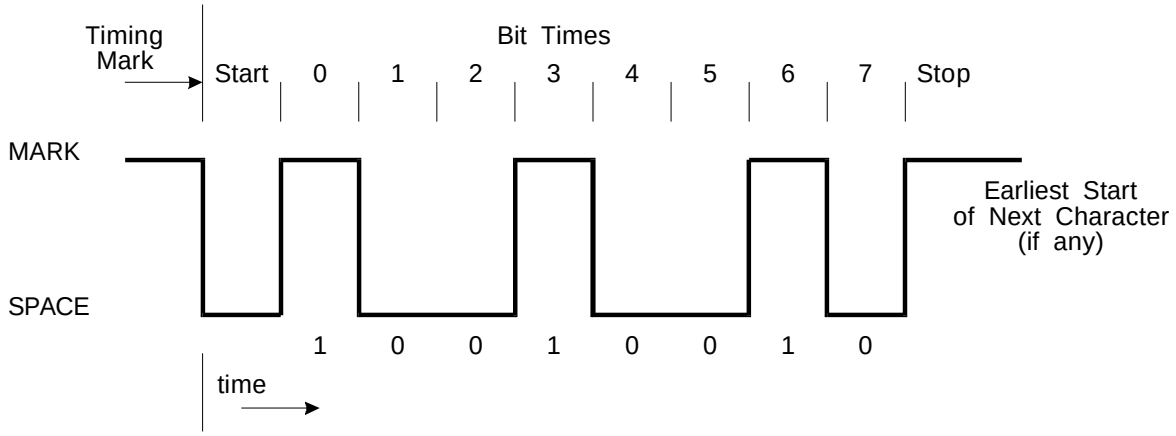
Finally we come to the signals. These are not part of the RS-232 standard and we go back to our three forms of communication.

Asynchronous Communications

Asynchronous communication occurs when "messages" (characters) are sent irregularly. It is also assumed, although not required by the terminology, that there is only one set of wires to carry the information and therefore each character must therefore contain its own timing information. The most popular format works in the following manner:

The line is initially defined to be in the MARK state which is also the state when there

is no modem link established. The start of a character is represented by the occurrence of a transition form MARK to SPACE and all timing is measured from this point. Time is now divided into "bit times" which must be agreed upon beforehand between sender and receiver. The bit time also happens to be the minimum time between edges which is the reciprocal of the "baud rate", the rate at which edges occur[4]. Thus on a 4800 baud line the bit time is 1/4800 secs.



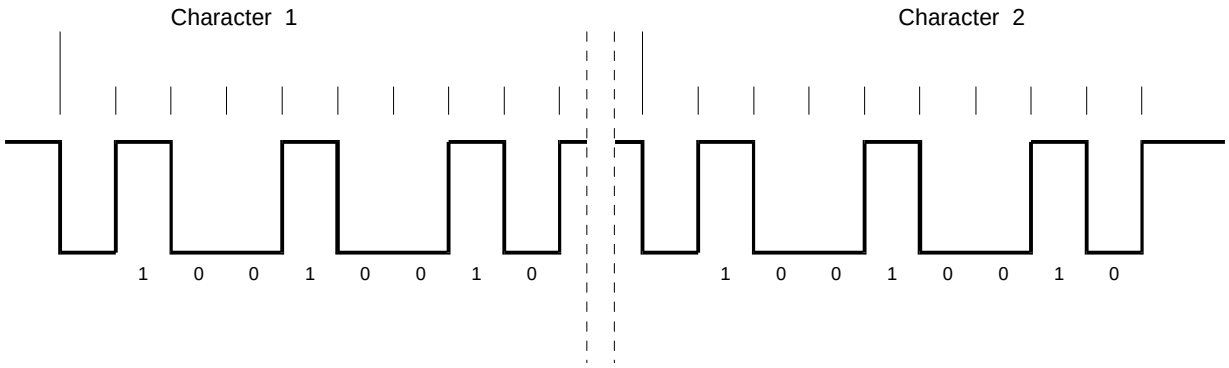
An Asynchronous Character Transmission

After one bit time - the start bit - the character is sent one bit at a time, a 1 being the MARK and a 0 being the SPACE, least significant bit first for an agreed number of bits.

After the data has finished the line must go to the MARK state in order to be ready for the next character. In order to ensure that this can happen and to give everybody a rest the minimum time between characters is also specified in terms of "stop bits" which may be fractional (e.g. 1.5 stop bits).

---

[4] This is the strict definition of "baud rate" - the rate at which edges appear in the system. It is often confused with "bit-rate" because the two have, in the past, been the same in many systems. However modern systems often have bit rates well in excess of the baud rate.



Successive Characters on an Asynchronous Line

Notice that there has to be considerable agreement between sender and receiver, and if you get lost there is a great deal of trouble to find yourself if characters are being sent end-to-end. There is also a problem with timing in that all times are measured from the start of the start bit and therefore a small change in the clock speed between sender and receiver endangers the last bit e.g if we nominally set our receiver equipment to check the line state in the middle of the bit time then on a 7-bit + parity system the start - parity centre time is 8.5 bit times. Now we can stray 0.5 bit times on an ideal system and still be correct (6% variation between sender and receiver - 3% each) but in practice we must allow for pulse distortion and the tolerances are even tighter - say 2% or better. Crystal controlled clocks of exactly the right frequency are a great help here, but people have been known to take short cuts and use slightly erroneous clocks which work fine on local equipment but won't run on modems.

By customary usage there are a fixed set of rates in use for asynchronous communications which are:

Rate (baud)	Usage	Rate (baud)	Usage
110	obselete teletype	4800	direct wire terminals
300	obselete modem	9600	direct wire terminals, fast modems
330	obselete modem	19200	short direct wire
600	obselete	38400	v .short direct wire
1200	slow modems	76800	fibre optics
2400	cheap modems	153600	fibre optics

Notice that nearly all these frequencies are binary divisions of the "magic frequency" of 2.4576MHz - a point which is often exploited in constructing the clocks for such links.

Flow Control

There is often a need to regulate the rate at which things are sent down an asynchronous line. There are two essential methods of doing this:

a) To use a proper communcations protocol such as the ENQ/ACK protocol discussed earlier or some alternative - software flow control.

b) To use some of the extra control signals from the system - hardware flow control.

Software Flow Control

Besides the protocols mentioned earlier, there is one very popular method colloquially known as "CTRLQ/CTRLS". This functions by having the receiver send a character 13(h) to halt the send and a 11(h) to resume. In fact these could be any characters at all - they are just single character aliases of STOP and RESUME.

There are many problems with this form of flow control and indeed it is somewhat surprising that it works at all. Some of the problems are:

No definition of the initial or current state. There is nothing to tell anybody that we are currently in a SEND state. We could assume that we are when we power up - but no-one will help us if we lose track after that! Conventional practice says that you should send a RESUME as soon as you are able to receive data after initialisation and any time that you think that you've been waiting rather a long time for some data.

No definition of reaction time. There is no guarantee of how long after you send a STOP command that the transmitting system will actually stop.

No error checking. If a STOP or RESUME gets lost nobody will do anything. If after the first STOP the transmitter doesn't stop - better send another STOP, and another, and another…..

Bidirectional control is also a problem if you are using the full character set as the control characters can appear in the data stream.

Hardware Flow Control

Two possible sets of control lines exist to use for flow control Clear-to-Send(CTS)/Request-to-Send(RTS) or (somewhat more commonly) Data-Termina-Ready(DTR)/Data-Set-Ready(DSR). These lines are used as STOP/GO lines and the required state is immediately apparent upon interrogating the current state of the lines. There is still somewhat of a problem with timing as none is defined, but this doesn't in practice seem to be too much of a problem. Bidirectional control is simple.

Thus in DTR/DSR flow control the DTE asserts the DTR line when it can accept data and de-asserts it when it cannot. Similarly the DCE uses the DSR line to indicate its availability for data reception.

Some advice - software flow control is very popular (saves wiring up all those pins) but hardware control is more reliable.

High Speed Modems

Modems have become technically very advanced and are now capable of very high

transmission speeds. It is often not clear how many "baud" a modem has or even how many bits-per-second it can transmit. This is due to a number of factors:

Phase modulation permits a modem to transmit more bits/sec than the baud rate

Adaptive Line Speed the modem can change the line speed according to conditions. Good lines will let it work faster, bad lines will slow it down and it will adapt as it goes.

Built-In Compression The data stream id compressed and expanded "on the fly" so that the data appearing at the ends of the DCEs apparently passed through the system at faster than the throughput rate.

Flow Control All this is only possible with flow control so that data doesn't suffer a "pile up" on the data highways.

As an example, the modem next to me on the desk will transmit an umcompressed ASCII text file at about 2,500 characters/sec with a published baud rate of 14,400. If I compress the file using an appropriate algorithm and then transmit the file, the rate drops to 1,350characters/sec. However the file size is often more than halved so it is actually faster to transmit compressed files. On the other hand it's time and bother to compress/expand them.....

Asynchronous Summary

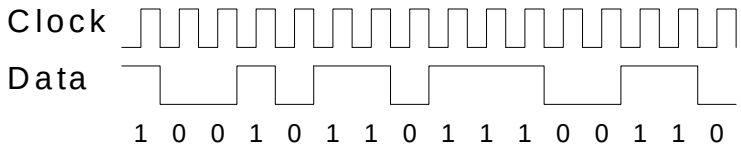
Having said all the above, despite being unpleasantly prone to interpretation problems and ambiguities, asynchronous communications using the above standards are the "bread-and-butter" of the computing industry for terminal handling.

Synchronous Communications

Synchronous communications implies that there is clock information in the data stream on a continuous basis, and therefore that the data stream is continuous. The clock may be carried either explicitly as a "clock line" or implicitly in the data stream itself.

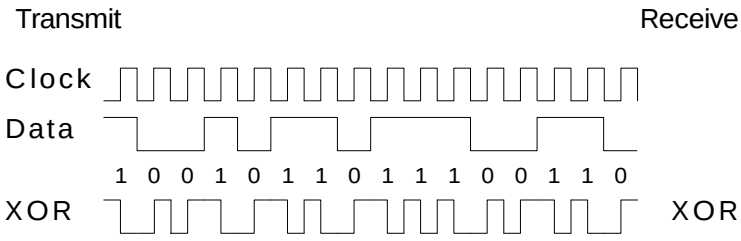
The simplest case to consider is that of the separate clock where the contents of each

data bit is defined as the data line state at an agreed transition (+ ve or -ve) of the clock[5] as shown in the accompanying diagram.



Separate Clock Synchronous Data

Since the transmitter provides the clock, the receiver is relieved of the chore of providing it and the only agreement required is on the range of allowed rates so that equipment has a sufficient time to function. However notice that the clock line has two transitions per bit time and therefore the line must be capable of supporting twice the bit rate as the "baud rate". Notice also that the clock runs along a different path to the data stream and therefore any differential delays in the system lead to problems if they cause a differential shift of more than 0.5 bit times.



Embedded Clock Data Transmission

This defect of synchronous communications (plus the expense of requiring two lines) can be overcome by the use of an "embedded clock" technique where the clock information is embedded into the data stream. Consider the case where we make an exclusive "OR"[6] operation between the clock and the data in the previous example. In this case we now have a situation where the "edge" of the data (positive or negative) carries the information and the extra edges between the "data edges" serve only to get things right for the next bit. Retrieving the data from the single stream is a matter of regenerating the clock and then regenerating the data.

This communication technique goes under a several names - Bi-phase and Manchester II being common ones.

---

[5] This is known as "Non Return to Zero" or NRZ encoding. The term comes from magnetic recording which has three states of magnetisation (UP, DOWN and NONE) when "Return To Zero" encoding is possible. In more complex designations it is NRZ-L.

[6] Exclusive OR is a logical operation with two input logicals. It produces TRUE if the two inputs are different and FALSE if they are the same. It can be considered as a logical comparator function.

Regeneration of the clock can be accomplished in the following manner:

Generate a very short pulse for every edge of the input data.

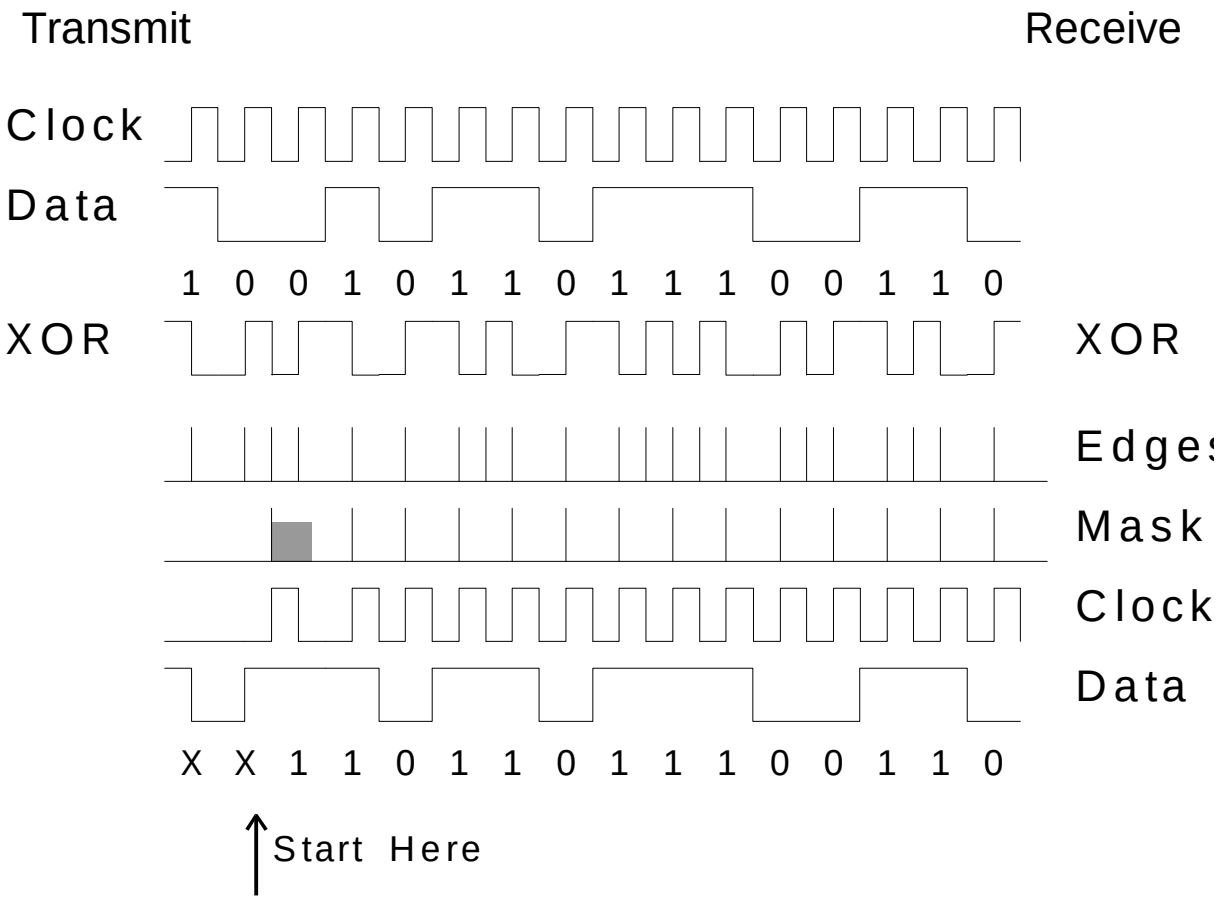
Starting at any pulse, ignore the pulse stream for 3/4 of a bit time then let through the next pulse which starts a 3/4 bit time blanking and so on.

Stretch the resulting pulses to 1/2 a bit time each - this is the clock.

XOR the recovered clock with the input to get the data stream out again.

If the correct edges are once found, then the system finds all the correct edges and all the correct data. What happens in the case of an error is bound up with a discussion of the blocking and transmission protocol. We will defer that discussion for a moment.

If we send a stream of "1"s or "0"s through the system we will generate a bi-phase waveform which is a square wave at twice the bit frequency - this cannot be used to set up the decoder because it is not possible to find the correct set of edges to "lock-on" to. However if the decoder is set up and makes no errors, the data stream will be correctly decoded. On the other hand if we send a sequence "101010.." through the system it generates a bi-phase stream at the bit frequency and the only edges in the system are the correct ones. Thus we see that "101010.." sequences are ideal for "training" the decoder and long runs of "1"s or "0"s will be a problem if the decoder makes a mistakes before the end of the sequence.



Recovering an Embedded Clock

Synchronisation

There is a more fundamental problem in synchronous communication and that is that the information on where the data starts has been lost and therefore the data stream is literally a "stream" of bits and is not divided into bytes, words or anything else. It is therefore necessary to "synchronise" or "train" the two ends to recognise where the boundaries are. I shall now talk about data "bytes" but the argument could equally well apply to any other length of data element.

Synchronisation involves the transmission and reception of some data pattern which can be recognised as being a

Synchronous Transmission in Blocks With "Sync"

synchronisation pattern and therefore allows the boundaries to be marked off from a point. If the link makes errors it will be necessary to re-synchronise before valid data can be received again. Some systems which transmit by blocks precede every block with a sync sequence to help out with this problem. The sync pattern has to be recognisable in any data stream but preferably should rarely occur by accident. To give an example, if the sync sequence is 8 bits long then in a random bit stream the e-fold[7] probability point that the sychronisation code will occur by mistake is about 256 bit times, for a 16 bit sync this drops to 65536 bit times, and for 32 bits to 4,294,967,296 or about 3.5 weeks at 2000 bits/sec. Thus a long sync sequence is preferable. Many systems use a repeated short sync and stuff repeats of the sync into any gaps in the data. There are in fact SYN characters in many codes such as ASCII and EBCDIC. Thus an ASCII system would stuff 16(h) on the line for sync or any data gaps and then ensure that no SYN characters were sent as data.

Setting up a data link and sending data requires that we perform a sequence of operations:

Bit synchronisation	Send 101010… which sets up the bit boundaries
Byte Synchronisation	transmit a sync code (or set of codes)
Send Data	
Send Error-checking	

Since the overhead on sending a block of information is quite high it is common to send data in large blocks. However the error rate interacts with the block length in a complex manner - generally higher error rates imply a smaller optimum block size and vice versa.

[7] the "e-fold" probability point is the point at which the probability of finding the sequence is $e^{-1} \approx 37\%$.