

## PART 4 DIGITAL METHODS

### Chapter 13 Digital Fundamentals

The remainder of this book will deal with digital methods in modern communications. We will build on the introduction to digital data in Chapter 3, but need some more introductory topics that will be used throughout the remaining chapters.

A recurrent theme in this chapter is that “timing is everything.” Most digital data transmission involves large — and possibly *huge* — numbers of bits. It is absolutely necessary to be able to identify each individual bit, and to know where it came from, and where it is to go. The method of doing this depends on the application.

#### Parallel Data Transfer

When binary data is moved from place to place, it is often moved in bytes. A byte can be transferred from place to place in one of two ways:

- Parallel transfer: all eight bits of a byte move at the same time along eight separate wires.
- Serial transfer: the eight bits travel on one wire, but in sequence (“serially”) one after another.

Actually, parallel data transfer requires more than eight wires. For example, the most common parallel connection is from a PC computer to a printer; this connection usually involves a 25-pin connector and cable:

- 8 wires carry the eight bits of data
- 1 wire carries the data-ready strobe, a signal that tells the printer that a byte is ready
- 1 wire carries a data-accepted signal from the printer back to the computer
- 1 wire carries an out-of-paper signal from the printer back to the computer
- 1 wire carries a busy signal telling the computer that the printer is busy
- 1 wire carries a ready signal, telling the computer that the printer is on line and ready to receive data.
- 12 wires connect the grounds of the computer and printer together, and perform some other functions.

The data-ready, data-accepted, out-of-paper, busy, and ready signals are called *handshaking* signals because they allow the computer and printer to agree on when and how fast to send data.

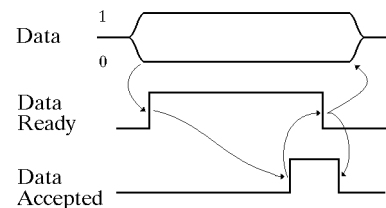


Fig. 13-1. Parallel data handshaking

Fig. 13-1 shows the timing of a parallel data transfer; we have simplified the situation here to just two handshaking signals — a data-ready signal, and a data-accepted signal. At the top, we have the actual 8-bit data. In real life, these eight bits would travel on eight separate wires, but there is no need to show so much detail, so this figure uses a common form of shorthand notation.

The data at the top is shown as a “hot dog” picture, which is supposed to show all eight bits in one. At the left, the hot-dog waveform is shown half-way between the 0 and 1 levels. This does not actually mean that the signals would be permitted to float somewhere between a high and a low level; instead, this is a shorthand notation which means that the actual signal level is not important at this time. Then the body of the hot dog expands up and down — this means that some of the eight lines may go up to a 1, while others may go down to a 0, depending on the precise bits being sent. Finally, after the data transfer, the hot dog returns back to its thin or “unimportant data” value.

Let’s use the words sender and receiver as we describe the actual sequence of events (follow the thin arrows in Fig. 13-1, which show which signal causes what):

- First, the sender sends out the parallel data.
- A short time later, when the parallel data has stabilized, it sends out a Data Ready signal to the receiver.
- The receiver takes some time to act on the received data, and then returns a Data Accepted signal.
- When the sender gets the Data Accepted signal from the receiver, it knows that both the data as well as the Data Ready signal have been received, so it first turns off the Data Ready signal, and then turns off the data itself.
- When the receiver sees the Data Ready signal go off, it knows that its own Data Accepted signal has been received, so it turns it off.

In the connection between a computer and its printer, there is a huge difference in their relative speeds — the

computer is typically much faster than the printer. Hence the handshaking provides a measure of *flow control* — controlling the speed at which data flows out of the computer to prevent the printer from being overwhelmed with data it cannot use. If the printer falls behind, it simply delays sending back the Data Accepted signal, thereby preventing the computer from proceeding. This means that the total time to send the data is variable — it depends on how fast the receiver can accept it.

A parallel connection can be quite fast since (1) all bits travel simultaneously, and (2) the handshaking signals allow the sender and receiver to communicate at a high speed, yet still slow down if one falls behind. On the other hand, the parallel connection requires many more wires, especially if you consider the need for a duplicate set of wires to carry data in the opposite direction. Hence parallel connections are only used for short distances.

All long-distance data transfer is therefore done through serial connections. In these, all data bits as well as handshaking signals travel along one wire in each direction (although there is also at least one additional ground or return wire.) Thus virtually all the techniques in the remainder of this book will involve serial transmission.

### Asynchronous Serial Data

The most common serial connection is known as RS-232; if you have a PC-compatible computer, there is most likely an RS-232 connector labeled COM1 on the back. Although it usually uses either a 25-pin or 9-pin connector, only two or three wires are absolutely necessary: one for a signal (or two if data travels in both directions), and one for a ground connection. This connector usually carries *asynchronous serial data*. Although asynchronous data and RS-232 do not have to go together — the term “asynchronous serial” applies to the data format, while RS-232 describes the actual connection hardware — they usually do and so we discuss them together.

For example, the ASCII code for the lower case letter *a* in a personal computer is 01100001. If you looked with an oscilloscope at the letter *a* carried on an RS-232 signal wire, you would see the waveform in Fig. 13-2.



Note this: an oscilloscope starts to display a signal from the left of the screen, and then moves right. In other words, the left side of its display occurred first, and the right side occurred last.

Although the code for the letter *a* is 01100001, the bits shown in Fig. 13-2 are 10000110, which is backward. That is because they really *are* sent backward.

When a binary number such as 01100001 is written down on paper, the rightmost bit (the *1* at the end of 01100001) is called the *least significant digit* or *LSD*, and the leftmost bit is called the *most significant digit* or *MSD*. But when this bit pattern is sent on an RS-232 link, the LSD is sent first, and the MSD is sent last. In the oscilloscope display, the LSD is thus at the left and the MSD is at the right.

RS-232 is most often used to carry ASCII characters, and so we will refer to the group of eight bits being sent as a character.



In an RS-232 circuit, the binary 1 is a negative voltage, labelled as  $-V$  in Fig. 13-2, while the 0 is a positive voltage labelled  $+V$ . The precise voltages are not specified, and could be anything from 3 volts up to 15 volts. Hence, in one system the voltages might be  $-5$  and  $+5$  volts, while in another they might be  $-12$  and  $+10$ , or whatever.

This is the opposite of what one often sees in other digital circuits, where a 0 is 0 volts and a 1 is  $+5$  volts or so. There is room for some confusion here, since many people wrongly think that a 1 has to be more positive than a 0, which of course is not the case here. To avoid that problem, many communications people therefore call the 1 signal a *mark*, while the 0 signal is called a *space*.

At the top of Fig. 13-2 is the notation “1 bit time” which shows the length of one bit. Within the byte, each bit has exactly the same length, which we call a *bit time*. The string of four zeroes in the middle, for example, is exactly four bit times long. Both the sender and the receiver must agree on the exact length of a bit so that, when a string of ones or zeroes arrives, the receiver can determine exactly how many bits there are in that string.

Once we know the length of one bit, we can calculate the maximum number of bits per second. For example, if each bit is  $1/300$  second, there could be a

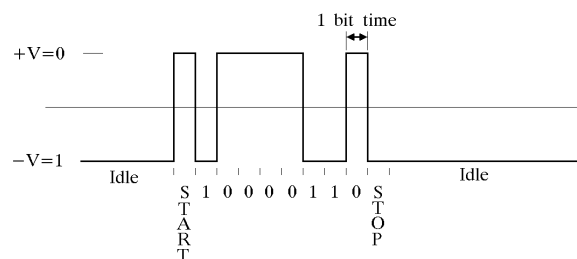


Fig. 13-2. A serial RS-232 letter "a"

maximum of 300 such bits sent per second. Thus the *bit-per-second* or *bps* rate is defined as

$$\text{bits-per-second} = \frac{1}{1 \text{ bit time}}$$

Since both the sender and receiver have to agree on the bit-time (and therefore the bps rate), there are certain bps rates which have become standardized over the years. These are 300, 600, 1200, 2400, 4800, 9600, etc. You can see the pattern in these. Incidentally, you often see the bps or bit-per-second rate referred to as the *baud rate*. This is not entirely correct, since baud rate has a different meaning from bps; still, it's a common use, and we might as well live with it. (When we get to discuss modems, we will see what the difference is.)

Although the timing of the bits within a character is very exact, the timing between characters is not. For instance, if the signal is coming from a keyboard, there might be long spaces between characters as the typist is searching for the next key. For that reason, this kind of serial data transmission is called *asynchronous*, meaning not synchronized. The bits are carefully timed (synchronized to a clock), but the characters are not.

Thus there has to be a way of telling the receiver when there is nothing being sent, and when the next character begins. The “nothing is being sent” condition is called an *idle*, shown as a 1 or mark signal in Fig. 13-2. Note that there can also be a 1 or mark signal inside a character, but that will generally be shorter.

The “character is starting” code is called a *start pulse*, and is always a 0, or space, which follows the idle. Thus a long mark (1) followed by a space (0) pulse marks the beginning of a new character.

Since the sender and receiver will always agree beforehand on the number of bits in the character (usually eight), they can count bits from the START pulse and figure out when the character is over. Hence they don't really need a stop signal. Nevertheless, there is always a *stop pulse* sent at the end, which is always a 1. (This is another of those “features” dating back to the early days of computers, when mechanical distributors much like the distributor in a car were used to convert to and from serial data in teletype machines. These distributors were run by a motor, and a clutch needed time to start and stop the distributor for every character. In fact, these systems often needed extra time, and so the STOP pulse was extra long. We still sometimes hear of “two STOP pulses”, which really just means a STOP pulse of double the normal length.)

When serial data is sent slowly, such as from a keyboard, the STOP pulse is usually followed by another idle signal of some unknown length. But when the data comes from a computer, it can come at maximum speed.

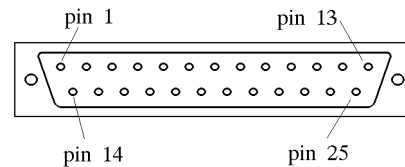


Fig. 13-3. The “official” DB-25 connector for RS-232

In that case, the STOP pulse might be immediately followed by another START pulse, with no idle between them. If you start watching such a data stream in the middle, things can get somewhat confusing since you can't tell which zeroes are START pulses, and which zeroes are data bits. To be really sure, you must go back to the last previous idle time, and start counting bits from there. Computers also often get confused — if you temporarily break the connection in a serial line you disturb the timing, and all the following data may be wrong until things slow down and the next idle reestablishes synchronization.

In any case, we now see that an eight-bit byte sent on a serial line actually takes a total of ten bits. The addition of the two START and STOP bits adds a 25% *overhead* which slows down the transmission.

## RS-232 and EIA-232

The term “asynchronous serial” applies to the data format we described above, while “RS-232” (or “EIA-232”, which is now more fashionable) describes a particular set of hardware and connections, used for sending serial data over distances up to about 50 feet.

RS-232 is the 232<sup>nd</sup> “registered standard,” originally issued in 1969 by the EIA, the Electronic Industries Association. The EIA publishes a number of such standards, designed to ensure that equipment made by different manufacturers can be used together. The RS-232 standard has gone through a number of revisions, with the latest (at the time of writing) being the RS-232-E version.

The early versions of the standard did not describe a specific connector, but the current versions specify a 25-pin connector dubbed the DB-25 connector, shown in Fig. 13-3. Nevertheless, many manufacturers use a 9-pin connector instead, and in practice there are two versions of RS-232: one that might be called the “de jure” version, which is the official version as described in the RS-232 documents, and one that can be called the “de facto” version, which is the way it actually exists in the real world.

TABLE 13-1. 25-PIN RS-232 SIGNALS

Pin no.	Short name	Source	Signal Description
1			Cable shield
2	TD	DTE	Transmitted data
3	RD	DCE	Received data
4	RTS	DTE	Request to send
5	CTS	DCE	Clear to send
6	DSR	DCE	DCE ready
7	GND		Signal ground
8	CD	DCE	Carrier detect
9			Unassigned
10			Unassigned
11			Unassigned
12		DCE	Secondary carrier detect
13		DCE	Secondary clear to send
14		DTE	Secondary transmitted data
15		DCE	Transmission signal element timing
16		DCE	Secondary received data
17		DCE	Receiver signal element timing
18		DTE	Local loopback
19		DTE	Secondary request to send
20	DTR	DTE	Data terminal ready
21		both	Remote loopback/quality detect
22	RI	DCE	Ring indicator
23		DTE	Data signal rate select
24		DTE	Transmit signal element timing
25		DCE	Test mode

### The official RS-232

The official RS-232 standard uses the 25-pin connector shown in Fig. 13-3; table 13-1 lists the signal names and functions on the pins.

The original RS-232 standard was designed for connecting computers or terminals — which are called *Data Terminal Equipment* or DTE — to communications equipment such as modems — which are called *Data Communications Equipment* or DCE. Each of the signals in Table 13-1 is thus labelled to show whether it comes from DTE or from DCE equipment. There are short names for all of the signals in the table, but we show only the most important ones.

Pins 2 and 3, the transmitted and received data pins, carry serial data in the two directions, while pins 1 and 7 provide the grounds. The rest of the signals provide various handshaking functions. After our earlier description of serial data is being used for long distances and wanting to minimize the number of wires, it may seem strange that there are so many handshaking signals. But

RS-232 is just one particular, and very popular, method of sending serial data over short distances.

Consider a typical case — a modem (DCE) connected to a computer being used as a terminal (DTE). The handshaking sequence would go like this:

- The modem (originally called a *data set*) sends DSR (Data Set Ready) to the computer to indicate that it is ready.
- The computer sends DTR (Data Terminal Ready) to the modem to say that it too is on.
- If the modem is off-line but detects an incoming call (the phone is ringing), it sends RI (Ring Indicator) to the computer, and answers the call.
- If the modem is online and has another modem at the other end of the telephone connection, it sends CD (Carrier Detect) to the computer, signifying that it is getting a carrier signal from the other modem.
- When the computer wants to send data to the modem, it turns on RTS (Request to Send), and the modem responds with CTS (Clear to Send). The RTS and CTS lines are most often used in cases where the modem is slower than the computer it is connected to, or where there are errors.

The other handshaking lines set the bit-per-second rate and do various testing and troubleshooting functions. They are especially useful with some very specialized, high-speed serial modems — or with very old ones!

The catch is that the vast majority of today's computers and modems do not need those specialized functions in RS-232, and in fact do their handshaking in a very different way.

### De-facto RS-232

The common RS-232 port actually used on today's computers is called COM1, and usually has the 9-pin connector in Fig. 13-4, not the 25-pin connector shown in Fig. 13-3. It uses the signals in Table 13-2. Only the most important handshaking lines are actually present.

TABLE 13-2. 9-PIN RS-232 SIGNALS

Pin no.	Short name	Source	Signal Description
1	CD	DCE	Carrier detect
2	RD	DCE	Received data
3	TD	DTE	Transmitted data
4	DTR	DTE	Data terminal ready
5	GND		Signal ground
6	DSR	DCE	DCE ready
7	RTS	DTE	Request to send
8	CTS	DCE	Clear to send
9	RI	DCE	Ring indicator

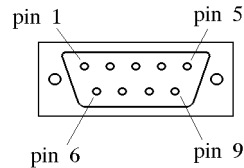


Fig. 13-4. The 9-pin RS-232 connector

The COM1 port today has a very limited number of uses. In the early days of personal computers, it was often used to connect to the printer, but today's printers almost always use either a parallel connection, or (very recently) a new serial connection called USB — the Universal Serial Bus.

The COM1 port can also be used to connect two computers together. Data can be passed back and forth by using terminal emulation programs, or two computers can even be networked together as if they were in a peer-to-peer network (terms which we will discuss in a much later chapter). But in this application, both computers play the part of DTE (Data Terminal Equipment), which generally means that their connectors do not match. For example, both computers would output data on pin 3, but listen on pin 2. A special cross-connect cable, called a *null modem*, has to be used to connect pin 2 on one computer to pin 3 on the other, etc.

Another use for the COM1 port is to connect to a modem. But instead of needing all the handshaking lines that old modems needed, today's modems handshake differently.

## Hayes Command Set

One of the first companies to manufacture modems specifically for personal computers was Hayes Microcomputer Products. Hayes invented and patented a special modem command method called the *Hayes Command Set*, which was later adapted by almost the entire modem industry. In this method, the computer and the modem communicate by sending special sequences of characters through the normal RD and TD wires.

Such a modem can be in one of two modes: either connected and talking to another modem at the other end of the telephone line, or not connected. If it is connected, it is said to be in *Connect Mode*; if not, then it is in *Command Mode*. The computer can force the modem to switch modes by sending codes on the TD line. For example, if the modem is in connect mode, the computer can make it go into command mode by waiting a few seconds and then sending three plus signs as in “+++” on the TD line.

Once the modem is in command mode, it will listen to commands sent by the computer, as long as the com-

puter prefixes the command with the letters “AT”, for “Attention!”. An entire booklet is usually needed to explain all the possible commands, but here are a few examples:

- ATZ = Reset the modem to its starting configuration. The modem will return the letters OK on the RD line if all goes OK
- ATH1 = Go off hook
- ATH0 = Go on hook
- ATDT1234567 = dial the number 1234567 using tone dialing

Similarly, if something happens on the telephone side, the modem tells the computer on the RD line what is going on, using plain English words. For example:

- RING RING = the modem senses an incoming call because the line is ringing
- CONNECT = the modem has dialed a number and connected to a modem there. A message like CONNECT 14400 might be used to identify the speed at which it is connected.
- BUSY = the modem has dialed a number, but it is busy
- NO DIAL TONE = the modem has gone off hook to dial, but there is no dial tone.

When a modem uses the Hayes Command Set language to communicate with the computer, there is no need for all the handshaking lines on the full 25-pin RS-232 connector; this is where the 9-pin connector is perfectly adequate. In fact, so-called *internal* modems may be mounted inside a computer and internally wired directly to its UART without any connector at all.

## The UART

Inside a computer, data is transferred in parallel. When asynchronous serial data is required, some device in the computer must do the conversion to and from serial. This is most often done by a special-purpose integrated circuit called the UART or Universal Asynchronous Receiver-Transmitter. (There is also synchronous data, which we will discuss later in this chapter. A USRT or USART handles synchronous conversions, but for now we discuss only the UART.) The UART is a general-purpose IC; some manufacturers make specific versions for certain applications and give them slightly different names, but we will use the name UART. In older systems, the UART was a separate IC; in modern systems, the UART function is often combined with other circuitry into a larger IC which handles many other functions as well.

The UART's job is complicated by the fact that digital signals are often corrupted in transmission, so they do not look like the nice square signals at the top of Fig. 13-5. The bottom wave in Fig. 13-5 shows an asynchro-

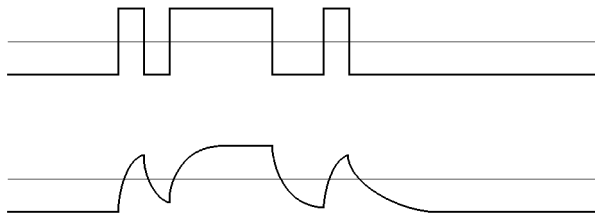


Fig. 13-5. Corrupted serial data

nous signal as it might look after being sent through a capacitive circuit; in practice, signals usually look even worse. The signal entering the UART receiver is therefore passed through a wave-shaping circuit which attempts to square up the signal (and which also usually converts RS-232 voltage levels into TTL-compatible voltages), but which also further messes up the exact timing.

Fig. 13-6 shows the block diagram of a basic UART. The IC consists of two relatively independent sections — a receiver that receives serial asynchronous data and converts it to parallel, and a transmitter which converts a parallel input into serial asynchronous data. There are also some control circuits (not shown in the figure) which count and control the number of bits, generate and check parity, and so on; we will ignore those here.

The inputs and outputs shown in Fig. 13-6 are pretty much what you'd expect — parallel inputs and outputs and the handshaking signals that go with them, serial inputs and outputs, error outputs, clock inputs, and some control inputs that are not shown. (These signals are generally at TTL-level voltages, so external voltage converters are needed to convert to and from RS-232 voltages.) Let's talk about the error and clock signals.

The receiver can detect three kinds of errors, and thus has three error outputs:

- A *parity error* signal if the received parity bit is not what the receiver is looking for
- A *framing error* if the STOP bit is wrong. This kind of error usually occurs if there is some timing error that placed the STOP bit at a different place from where the receiver expects it
- An *over-run error* occurs if incoming serial data is coming in faster than the device connected to the parallel output is taking them out of the UART.

In a basic UART, the receiver and transmitter can run at different bit-per-second rates (that is not true in some of the special-purpose IC's from some manufacturers) so there are separate receive and transmit clock inputs. In a basic UART, these run at 16 times the serial bps rate — for example, a UART running at 300 bps would use a

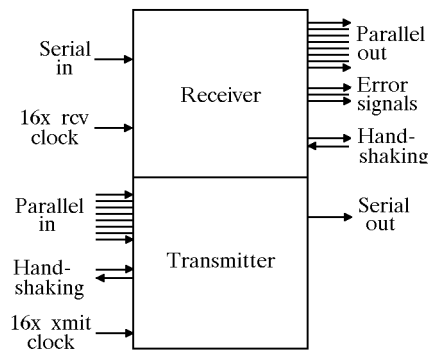


Fig. 13-6. UART block diagram

clock running at 4800 pulses per second, sixteen times faster. Thus the clocks are labelled “16x clock”. (In some UARTs, a factor other than 16 might be used.)

The reason for the higher-speed clock has to do with the receiver. By the time the signal actually arrives at a receiver, it might be severely distorted and its timing could be very strange. The UART can't therefore rely on a clean serial input signal. It therefore works like this:

(1) When a long 1 or mark signal arrives, assume it is an idle between characters, and just wait.

(2) When a 0 or space signal arrives, this could be the beginning of a START pulse, but it might also be a noise spike. So don't jump to conclusions — wait  $\frac{1}{2}$  of a bit time, and check it again. If it is *not* still a 0, then it was probably a noise pulse, so go back to step (1) and wait some more. If it *is* a 0, then continue to step (3).

(3) At this point, we are somewhere in the middle of the START pulse (remember the  $\frac{1}{2}$ -bit-time delay from the beginning of it).

(4) Wait 1 bit time until we are in the middle of the next bit, and grab this bit.

(5) Repeat step (4) until we get all the bits of the character, check for errors, and if OK then send it out the parallel output.

So the receive portion of the UART must be able to detect the beginning of a START pulse, wait half of a bit time, and then time each of the remaining bits as they arrive. A receive clock oscillator sets the basic clock timing, but remember that the incoming serial data is *asynchronous* — the START pulse can arrive at any time. There is no way to exactly synchronize a local clock to these randomly arriving inputs.

That's where the 16x clock comes into the picture. The clock generates 16 pulses for each bit time of the incoming signal, as in Fig. 13-7. When the UART detects the beginning of a START pulse, it waits 8 of those clock pulses, which brings it to the approximate middle

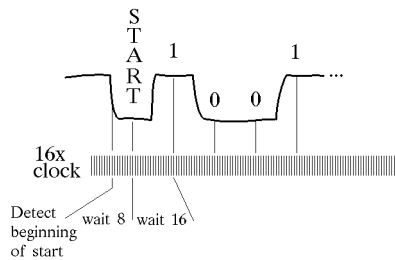


Fig. 13-7. Correct UART receive timing with 16x clock

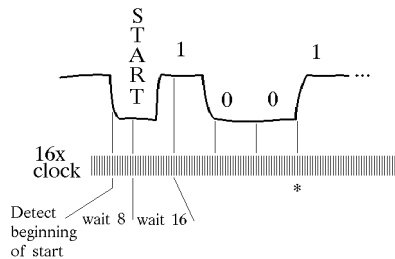


Fig. 13-8. Bad UART timing with the wrong bps rate

of the START pulse. For each additional incoming bit, it waits for 16 clock pulses, which again brings it to the approximate middle of the next bit. Since the receive clock is not synchronized to the incoming data, there will be a slight error in the timing, but the error should not exceed one cycle of the 16x clock.



The UART's 16x clock sets the speed at which the UART will receive data; a 16x clock is also used to set its transmit speed. Our discussion above was actually a bit oversimplified. Early UARTs all used 16x clocks; changing the bit-per-second rate required changing the clock speed, so a device providing several different bps rates required different circuits for providing the clock signal.

Modern UARTs use a higher clock speed, but use programmable internal counters to count off different numbers of pulses depending on the desired bps rate. For example, one common device uses a clock of 1.8432 MHz. At 300 bps, for example, 1.8432 MHz is exactly 6144 times 300, so the UART counts off exactly 6144 clock pulses for a one-bit delay, or 3072 pulses for a half-bit delay. The same clock would be used for 9600 bps, but now the UART would count off 192 pulses for a one-bit delay, or 96 pulses for a half-bit delay. In this way, one clock oscillator can be used for a variety of bps rates, at the expense of a slightly more complicated

UART. In this case, the clock would no longer be called a 16X clock at all, yet it still provides the same function.

So let's not worry about the exact structure of the system; let us just understand that the sender and the receiver must both agree on a common bps rate, and use appropriate clock sources to transmit and receive at the same rate.



### Timing errors

We should, though, look at another possible error — what if the receive clock is slightly fast or slightly slow? For example, suppose the incoming signal is supposed to be at 300 bps, but is at 270 bps instead, while our receive clock is still at exactly  $16 \times 300$  or 4800 pulses per second. Since there are fewer bits per second, each bit is slightly longer than expected. Hence the receiver will grab each bit just a moment too soon. The errors accumulate, so the farther to the right you look, the worse it gets. Fig. 13-8 shows the result, and the \* marks the first error where the receiver grabs the wrong bit. From then on, it's all downhill, as each bit gets worse and worse.

So the question becomes this: How much of an error can you tolerate before errors occur?

A complete ASCII character sent serially takes 10 bits — 8 bits for the character, plus a START bit and a STOP bit. A slippage of  $\frac{1}{2}$  bit time on the tenth bit will produce an error, so the maximum allowable difference between the transmitter's clock and the receiver's clock is  $\frac{1}{2}$  bit time out of 10, which is a 5% error.

Assuming that both the transmitter and the receiver might be off but in opposite directions, the maximum error for each should be less than  $2\frac{1}{2}\%$ ; but since pulse distortion might also introduce further errors, a more practical limit on timing errors would probably be around 1% or even less. In other words, the actual clock accuracy in a UART should be at least 5 or 10 times better than the 5% that produces a guaranteed error.

Using quartz crystals for frequency reference makes it easy and cheap to achieve this kind of accuracy, so this is not a problem with asynchronous data transmission. The advantage here is that, even if an error occurs in timing the bits of one character, the timing starts all over with the START bit of the next character. In other words, errors accumulate within one character, but then the slate is wiped clean and it starts all over in the next character.

## Synchronous Serial Data

The asynchronous serial method we've discussed so far has the advantage of being simple, but it also has a number of disadvantages. The primary one is that it is inefficient — not only are there two overhead bits for every eight data bits (which wastes time), but the parity bit (if used) adds still another wasted bit that doesn't really do a good enough job of detecting errors. For every eight data bits, we have three extra, useless bits. Synchronous data transmission is a way around that.

In the synchronous method, many bytes of data, possibly thousands or millions, are sent one right after another, without separating them by STOP and START bits. By using a more thorough error detection scheme, synchronous transmission reduces the chance of errors. By transmitting more bits in a row, it reduces the number of extra overhead bits, and thus increases the efficiency. But timing now becomes very important because a very large number of bits is sent in a row, and even a slight error in timing them could cause a miscount. For example, suppose a string of a million bytes (eight million bits) is sent. To achieve an error of less than  $\frac{1}{2}$  bit time out of 8,000,000 bits requires an accuracy of

$$\frac{\frac{1}{2}}{8,000,000} \times 100\% = 0.000006\%$$

Even with quartz crystals, achieving this kind of accuracy is no trivial matter.

## Synchronizing transmitter and receiver clocks

There are really only two solutions when such high accuracy is needed — either use two very precise atomic clocks, or use the same clock to time both the sender and receiver. But these can be implemented in many different combinations:

- Use two very precise clocks, either located at the sender and receiver, or located at some other place.

Fortunately, precise atomic clocks are no longer as expensive as they once were. Not so long ago, the entire high-speed communications system linking the U. S. telephone network was run from a single Bell System atomic clock (called a *Stratum 1* clock) located in Missouri, near the geographic center of the United States. Now, cheaper atomic clocks exist throughout the country; another alternative is to use a GPS receiver to receive atomic clock signals from the 24 U.S. Defense Department Global Positioning System satellites circling the globe.

- (2) Use one clock to time both ends of the connection. In this case, the clock could be at some

central place (such as the Stratum 1 clock mentioned above), or else at either end of the connection.

When the same clock is used by both the sender and the receiver, it does not generally need to be as accurate. As long as both the sender and the receiver are running at the same bps rate, that is generally enough — except in those cases where a series of circuits are all tied together, in which case it is very useful if they are all running at the same speed.

The problem is how to get that clock signal to both places at the same time without using extra wires. One solution is to use self-clocking codes — methods of sending data and timing or clock pulses on the same wire at the same time.

## Self-Clocking codes

There are many ways of representing bits on a serial line, and Fig. 13-9 shows a number, though not all. Let's run down the list:

- *Unipolar NRZ* is the simplest. This is what you would see in simple TTL circuitry, where you use one voltage (such as +5 volts) for a ONE, and another voltage (such as 0 volts) for a ZERO. There are some variations, such as using the positive voltage for a ZERO, or using negative voltages instead of positive voltages. (The word unipolar means there is only one polarity of voltage, and NRZ means *non-return to zero*.)
- *Bipolar NRZ* looks almost the same as unipolar NRZ, except that both positive and negative voltages are used. In this particular case, the ONE signal is positive, while the ZERO is negative.

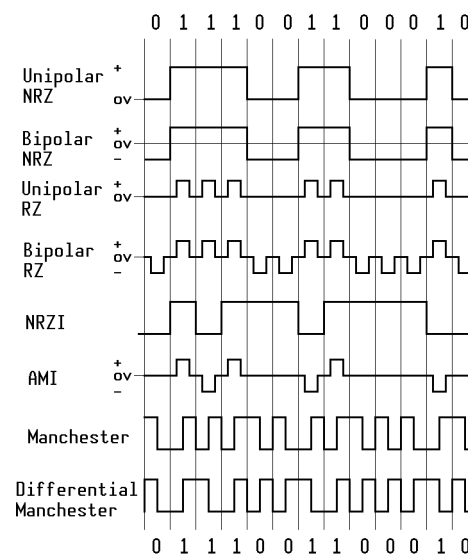


Fig. 13-9. Serial data encoding methods

(The word bipolar means there are two voltage polarities used.) An RS-232 signal is bipolar NRZ, but with the variation that the ONE is negative and ZERO is positive. The two NRZ forms are also sometimes called NRZ-L, with the L meaning that bits have a particular voltage *Level*.

- *Unipolar RZ* and *Bipolar RZ* are similar to the two NRZ versions in using one and two polarity voltages, respectively, but RZ means return to zero. That is, the voltage swings to plus or minus for a bit, but then returns back to zero volts between bits.
- *NRZI* is *Non-Return-to-Zero-Invert*. Like plain NRZ-L, it can be bipolar or unipolar, but instead of using a particular voltage level for all digital ONE signals, it uses a voltage *change* for each ONE bit. Whenever there is a ONE bit, the voltage changes; when there is a ZERO bit, the voltage stays the same.
- *AMI* or *Alternate Mark Inversion* is bipolar, and uses three voltage levels, but this time ZERO bits are encoded as 0 volts, while ONE bits are carried as alternating positive and negative voltages. That is, if one ONE bit is positive, the next ONE will be negative, the next ONE is positive, and so on.
- *Manchester* code seems complicated, but is really quite straightforward. It uses just two voltage levels, but in the middle of each bit-time is a voltage change. That change goes up (toward positive) if that bit-time carries a ONE bit, and goes down (toward negative) if it carries a ZERO bit. Between bits, the signal may go up or down, as necessary, to properly connect the two bits together. For example, if one bit is supposed to go low-to-high (and thus ends high) and the next one also goes low-to-high (and thus begins low), there has to be a voltage change to connect the ending high of the first bit to the beginning low of the second.
- *Differential Manchester* also has a change in the middle of every bit time, but these changes are slightly more complicated than those of regular Manchester code. The general rule is as follows: the change in the middle of a bit time is (a) the same as the previous bit for a ZERO, but (b) opposite to the previous bit for a ONE. In our example, the first bit is a ZERO, and its middle happens to go down. The next bit is a ONE, and so its middle goes up, opposite to the previous bit. (Despite its complexity, there is no substantial advantage to differential Manchester as com-

pared with Manchester; we are discussing it simply because it is used.)

Why all of these different schemes (and there are more)? All of them are perfectly capable of sending a string of bits from one place to another, so let's discuss some of their features, but first a short

## DETOUR

We discussed phase-locked loops in Chapter 10. We pointed out that the PLL has a voltage-controlled oscillator (VCO) whose frequency is locked to an incoming reference signal. As the frequency of the incoming signal changes, so does the frequency from the VCO.

We also mentioned that the PLL circuit contains a loop filter — but never really explained what this filter does. Without going into the mathematics, let us say that the filter provides a delay and averaging function, which allows the VCO frequency to stay constant for some time even if the incoming reference frequency changes waveshape or even temporarily disappears.

This means that you can design a PLL circuit which will input one of the data streams from Fig. 13-9, and lock itself to the bit frequency of the signal. That is, the PLL's VCO will set itself to the same frequency as the bit rate of the incoming signal.

A properly-designed PLL can therefore be used to recover a timing clock signal from a serial data stream, and use it to time the receiver circuit. But the loop filter only provides a delay and averaging for a short time — if the input signal were to disappear for a long time, or if it should drastically change, then the PLL would become unlocked and drift off to the wrong frequency.

## END OF DETOUR

Let us now return to compare some of the encoding schemes in Fig. 13-9.

Both versions of NRZ could be used to synchronize a PLL timing clock generator — as long as we have continuous alternating ONES and ZEROS. But what would happen if the data contained a long string of 00000000... or 11111111... bits? Both of these conditions would result in a relatively long period of constant voltage (high or low), so that the PLL clock generator would unlock. The receiver would then lose timing and start making errors.

NRZI and AMI have the same problem with a long string of 00000000.. bits. But they have some advantages over plain NRZ. NRZI is somewhat easier to detect in the presence of noise, since it relies on changes rather than specific voltage levels, and changes are easier to spot in noisy signals. AMI has the advantage of

involving lower frequency components, so it works better on cables that were never intended for digital data. It also has some built-in error detection: alternate ones (marks) should be opposite polarity; if two ones ever appear that have the same polarity, this indicates an error (called a *bipolar violation* by telephone people.)

RZ and Manchester codes, on the other hand, are very busy — every bit has something happening, regardless of value. No matter what digital data is being sent, the signal never stops moving. Hence a PLL would always have some reference signal, and would never lose lock. In fact, the voltage changes in Manchester code can be converted to clock pulses even without using a PLL.

This “busy-ness” is better for the circuit which attempts to recover clock signals from the data, but it also means that the bandwidth required is larger. You can see this from Fig. 13-9 — NRZ, NRZI, and AMI vary fairly slowly, while RZ and Manchester codes have many rapid changes.

Finally, bipolar RZ and AMI, by needing three separate voltage levels, are difficult to use in places where only two levels are available. For example, in an optical fiber, we have light or no light — we do not have “positive light” and “negative light”. Hence bipolar RZ and AMI are not suitable.

Each of these various encoding schemes thus has very specific uses. For example, the various forms of NRZ are used in simple, inexpensive circuits such as RS-232 cables. AMI is used for T1 carrier circuits in telephone networks. The superior clocking ability of Manchester code (or a similar code called Miller code) would be useful for recording data on mechanical media. For example, as a floppy disk turns in the drive, various friction forces make the speed very irregular, so the data clock speed varies a lot. Using Manchester (or Miller) code, the floppy drive circuitry would be better able to synchronize to the data because every bit essentially carries its own clock pulse.

## Bandwidth

In previous chapters, we discussed the bandwidth needed for analog signals. What about the bandwidth for digital data?

Although digital signals don’t exactly consist of square waves, they look close enough to square waves that we suspect that the bandwidth is similar. That is, we should expect something resembling a fundamental frequency, plus a large number of harmonics. Although we haven’t specifically said so before, a general rule of thumb is that the more signals change, or the faster they change — the more edges they have, the steeper they are, or the more corners they have — the higher the fre-

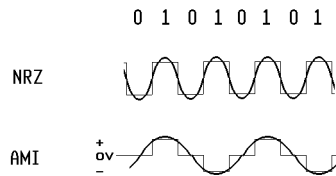


Fig. 13-10. Low-pass-filtered NRZ and AMI

quencies in that signal. This almost certainly means that there will be a large number of harmonics, and so it implies that we need lots of bandwidth.

Looking at Fig. 13-9, we therefore expect RZ, and especially bipolar RZ, to require a higher bandwidth than NRZ. Same for Manchester coding.

That is generally true; fortunately it’s not as bad as it seems. Even if a digital signal becomes all distorted, it may still be possible to read it without errors as long as the ones and zeroes do not get totally confused with each other.

Fig. 13-10 is an interesting example, which shows the bit pattern 01010101 encoded in both NRZ and AMI. In each case, the light waveform is the original “by the book” waveform, while the dark waveform superimposed over it shows how the signal might look if it were passed through a low-bandwidth circuit that lets through the fundamental, but not the harmonics. Clearly the fundamental frequency of the AMI signal is lower than that of the NRZ signal — at least for this particular bit pattern.

Because AMI tends to have lower bandwidth requirements than some of the other codes, it has been chosen by the telephone industry for use in T1 carrier systems (which we will discuss later.) AMI is ideally suited for sending T1 digital signals through old copper cables which were originally intended for analog telephone signals.