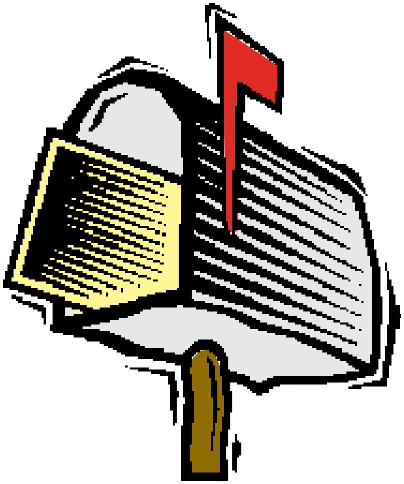
Letters, we get letters

But *how* we get them—aye, there's the rub



I'm really not that old, but I did once own a 300 bit-per-second (bps) modem. Back in those days, every bit counted. Because of this, many of the bits (no pun) and pieces of today's Internet reflect some of the odd standards required then to maximize every ounce of data transmission.

The best example of this is E-mail. Way back when, it was nigh-on impossible to send a program or a "binary file" (anything but pure text) over the Internet's E-mail system without some funky maneuvering. These limitations don't (in general) apply now, but the standards that were set then are still followed today.

Cutting your digital teeth

If you use Eudora, a popular POP (Post Office Protocol) mail client from Qualcomm, you've seen the terms UUEncode, Binhex, Apple Single, and Apple Double—and probably not known what they meant. Even if you use something besides Eudora, you've encountered one or more of these, though the last three are best-known in the Apple Macintosh community.

In the days of those old 300-bps modems (or the even-slower 110-bps modems—oh, yeah, they really existed), the protocols that drove modems had to do lots of error detection and "framing." These two items still play a principal role in the workings of all data communication.

Let's review basic computing, though, before I stick your head in this data structure. As you probably know, a bit is a single binary digit; it's either 0 or 1, with no values in between. A byte is made up of eight bits (some computers actually use different sets of bits than bytes, but typically data are transferred in byte-sized nuggets).

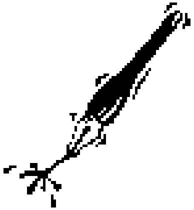
Since each position (remember, there are eight of these) in a byte is a bit that can represent one of two numbers, the eight bits in combination can represent 2 to the eighth power numbers, or a total of 256 different values. So an eight-bit byte can represent the numbers from 0 to 255.

Sometimes, as you'll see, you might use more bits or fewer to transmit a byte.

Errors and parity

Error detection (as opposed to correction) is the ability of a piece of data to contain information about itself that allows the recipient to check that the integrity of the data is intact. That is, if you send, say, seven bits, you want to know that the seven bits that left are the same ones that arrived—since they might not be, given noise on the line, static, machine burps, whatever.

Error detection is one of the cornerstones of information theory—the field whose practitioners spend a lot of time thinking about information transmission and signal (data)-to-noise ratios. Error *correction* is a higher-level idea; once an error has been detected, correction allows you to recreate the



information. For example, on all CD-ROMs, several bits get added to every byte to essentially allow the byte to be recreated even if part of it is unreadable or doesn't check out correctly.

Error detection is done through checksums and parity. Checksums are complex animals: they're the result of doing an equation on an entire set of data, such as a file you're sending. The equation derives a number based on all the data in the file. When the file's received on the other end, the machine there runs the same equation and checks the result. If it's off, the file was corrupted in transmission.

Parity is a simpler concept: you add the number of bits set to "1" in each byte you're sending. With "even" parity, if the number of bits set to "1" is odd, the parity bit is set to "1," making the total number of bits set to "1" an even number; otherwise, the parity bit is set to "0" (preserving an even number of bits set to "1"). Odd parity is the opposite of this.

"Framing" simply involves sticking a consistent signal (piece of data) on both ends of a byte. If you're sending 8 bits (7 data bits with 1 parity bit), a zero, called a "start bit," is automatically placed at the beginning, and you put two zeroes at the end—these are called "stop bits"—for a total frame of 11 bits. This arrangement is often referred to in terminal programs—programs used to dial into remote computers, such as UNIX systems—as "7-E-2," for seven data bits, even parity, two stop bits.

Framing is important because, if things get out of whack, the two modems involved will know that every eight bits will have three zeros between them (at least, in the scenario above). This allows them, for instance, to resynchronize after someone picks up the phone while someone else is in midtransmission.

ASCII and binary

All this brings us to the modern era. As modems improved in speed and technology, and phone lines went more and more digital behind the scenes, so that noise level went down, more data could be transferred more quickly and accurately, and less framing and error correction was necessary. Today's modems typically can support the 8-N-1 arrangement, which has eight data bits, no parity at all, and one stop bit.

You're probably asking about the seven-bits-versus-eight issue at this point. Seven bits can represent 2 to the seventh-power numbers, or the range from 0 to 127. It happens—by design, not accident—that, in those seven bits, you can encode the entire Roman alphabet, including most characters necessary for all major languages that use that alphabet; the figures zero through nine; punctuation; and 32 control characters that are really remnants of the days of workstation terminals.

So you don't really need that eighth bit to send pure text—and that's how this all starts to make some sense. Eight bits are required to transfer programs, because instruction codes and data for computer applications may be any number between 0 and 255. Seven-bit data are called ASCII (American standard code for information interchange) data, for the set of characters between 0 and 127, the ASCII set; eight-bit data are "binary," a misnomer that nevertheless means that the data employ values between 0 and 255.

But in the old days, when this was all codified, people weren't really transferring programs and other binary data between machines. They were using modems to type remotely on mainframes and minicomputers using "terminal emulation" programs. These programs allowed early home computers to dial up a mainframe and interact with it just as if they were dumb terminals (no CPU, you see, to do complex tasks locally; all the computation is done on the mainframe).

It was also certainly more efficient to send 10-bit bytes (7 data) instead of 11-bit bytes (8 data) when you could only transmit 110 bits per second! Sending an extra bit would have made you 10 percent slower—and at those speeds, you could watch the computer type at you.

When the faster modems were introduced, packing more data into the same phone lines, new protocols allowed the transmission of eight-bit, "binary" data. But by then, the rest of the system—especially its orientation toward seven-bit data—was cemented in place.

What the heck is UUEncode?

We're near the end of our journey now! So let's pretend you were one of those pioneers, and you had eight-bit data, such as a program, that you needed to transmit at the stunning speed of 300 bps to some remote colleague. How could you do this? Obviously, if you had a way of representing the eight-bit data in seven-bit chunks, you could still get the data across. But your colleague would need a program at the other end to decode this information and turn it back into binary data.

Thus we get UUencoding and UUdecoding, or Unix-to-Unix encoding and decoding. Unix to Unix was the idea of transferring data from one machine to another, both of them presumably running the Unix operating system. The encoding took a binary program in, and produced a seven-bit ASCII file that could be transmitted over modems using E-mail programs or other methods. On the other end, the recipient would run the decoder and turn the file back into pure binary.

(A related protocol developed in the Macintosh community is the Binhex format, which converts binary data to hexadecimal code—a base-16 system number format that uses the digits 0 through 9 and the letters A through F to represent the numbers from 0 through 15. It's not terribly efficient, but it was well developed and deployed among Mac users, who were without access to UUen/decoders—and the principal is identical.)

You're probably getting the picture: By the time modem protocols improved to allow binary transmission, the mail systems—and notably the underlying software that ran the modems connecting these systems—had all been built around sending and receiving only seven-bit ASCII ! So even though the world had sped past, many systems still couldn't deal with eight-bit data.

Even for the last several years, when most mail systems have been updated to deal with binary data, there's enough other incompatibility across systems, especially between commercial mail programs and their gateways, so that sending files in UUencoded format still is often the *only* way to get data through. Typically, the gateway will do the work of en- and decoding the data automatically, so the users sending data to each other don't have to run separate encoders and decoders.

(As noted above, Qualcomm's Eudora supports several methods of attaching files: UUencoding and Binhex, the two major ASCII encoding methods—both are included in the Pro version—and Apple Single and Apple Double. The Apple Single method encodes Macintosh files, which have special resource forks containing icons and other bits, into a format that allows them to be reassembled on the other side, while Apple Double compresses the file before its transmission, saving time in uploading and downloading.)

Out of the frame

As more and more people use programs like Eudora, or even America Online's E-mail system, we should see less and less of the old ASCII encoding. Two Eudora users sending to each other can choose any method in common without thinking about it. America Online's mail system has been able to automatically convert files attached in AOL when they're sent out to the

Internet, and decode most files coming in, too.

But, even though today's modems are a hundred times faster than those I cut my teeth on, and today's networks have literally thousands of times more capacity, from the point of view of some programs, E-mail is still basically a little straw sucking sand from one desert to another.