By Lewis F. McIntyre, *KB6IC*
3711 Gayle Avenue
Omaha, Nebraska 68123

ERROR CORRECTION IN DATA TRANSMISSION

Using Hamming codes to detect and correct errors in digital transmissions

HF communications techniques for hams have undergone a dramatic change over the past ten years. Electro-mechanical ASRs have given way to computer terminals, and Baudot has lost some of its popularity to ASCII, AMTOR, and packet. Data rates have increased from the venerable 45 baud to 300. But a "quick" packet exchange under less than ideal conditions proves that error detection and correction codes haven't kept up with these changes. In fact, packet and AMTOR use only error detection codes – requesting repeats when errors are found. AMTOR is effective, but slow. Under the best of conditions, its efficiency level is 50 percent. Packet, too, can quickly bog down on a noisy circuit because of its higher data rate.

For ordinary chitchat, straight RTTY is hard to beat. It's too bad the computer can't "fill in the blanks" on a few hits the way a human operator can, or can it?

Actually, your computer can fill in the blanks. Hamming codes, developed by Dr. Robert Hamming almost 40 years ago,[1] let computers detect and correct errors in digital transmissions. These codes are used in many applications. For instance, Hamming codes are used to ensure data integrity in the memory portion of the new VHSIC chips.

I'd like to introduce you to some Hamming code basics, and share some look-up tables and ideas for future development. I'll also analyze how these codes might be used to improve packet and AMTOR link performance.

How they work

Hamming codes can correct single-bit transmission errors. The mathematical process involved is quite complicated, so I'll skip the theory for now and go directly to an example.

Say you want to encode a four-bit data word into a seven-bit word called a "Hamming sequence." This is a 7/4 Hamming code (four data bits and three parity bits, for total of seven transmitted bits) which can correct single errors and detect two-bit errors in a received word. To see how this is done, pick a number between 0 and 15. Let's try 4. In **Table 1**, the encode table, find the number's Hex value (04H). Follow along the line to find the opposite value – 100 1100, or 4CH. This is the Hamming sequence you'll transmit. After you receive that sequence, decode it using **Table 2** – the decode table. The answer is 04H. Now simulate noise by changing any one of the seven received bits to its opposite value. Try making the least significant bit (LSB) a 1. This makes the received sequence 100 1101, or 4DH. Look up this Hamming sequence in **Table 2**, and read the decoded value. You'll find it's still 04H. Change any other bit, and you'll still obtain the correct value, 04H, from **Table 2**.

Why? The answer is obvious if you look at the decode table (**Table 2**). This table is eight times larger than the encode table (**Table 1**), because the decoded value of 04H (and all 15 other decoded answers) is decoded at eight entries. It's decoded once at the "no errors" entry of 100 1100 (marked with the asterisk), and seven times at 100 1101, 100 1110, 100 1000, 100 0100, 101 1100, 110 1100, and 000 1100 (all of the seven single-bit error positions possible).

Now change two bits. Make the sequence 100 1111. Your answer will be 07H, with errors detected. Two-bit errors will always give the wrong answer, but will never decode as a "no errors" entry marked with an asterisk. Depending on the word, three or four bits out of the seven sent would have to be changed for that to happen.

It's a bit harder to create a Hamming sequence[2] When doing so, you need to get into the mathematical part of the process. But because it doesn't really come into play once a look-up table sequence has been defined, I'll hold off on the theory once again. At this point, I'd rather pique your interest with some practical HF applications for this encode/decode scheme.

Data Word	Encoded Hamming Sequence Binary	HEX
00H	000 0000	(00H)
01H	110 1001	(69H)
02H	010 1010	(2AH)
03H	100 0011	(43H)
04H	100 1100	(4CH)
05H	010 0101	(25H)
06H	110 0110	(66H)
07H	000 1111	(0FH)
08H	111 0000	(70H)
09H	001 1001	(19H)
0AH	101 1010	(5AH)
0BH	011 0011	(33H)
0CH	011 1100	(3CH)
0DH	101 0101	(55H)
0EH	001 0110	(16H)
0FH	111 1111	(7FH)

Table 1. Encode table for 7/4 Hamming sequences.

Received Hamming Sequence	Data Word	Received Hamming Sequence	Data Word	Received Hamming Sequence	Data Word	Received Hamming Sequence	Data Word
000 0000	00H*	010 0000	00H	100 0000	00H	110 0000	08H
000 0001	00H	010 0001	05H	100 0001	03H	110 0001	01H
000 0010	00H	010 0010	02H	100 0010	03H	110 0010	06H
000 0011	03H	010 0011	0BH	100 0011	03H*	110 0011	03H
000 0100	00H	010 0100	05H	100 0100	04H	110 0100	06H
000 0101	05H	010 0101	05H*	100 0101	0DH	110 0101	05H
000 0110	0EH	010 0110	06H	100 0110	06H	110 0110	06H*
000 0111	07H	010 0111	05H	100 0111	03H	110 0111	06H
000 1000	00H	010 1000	02H	100 1000	04H	110 1000	01H
000 1001	09H	010 1001	01H	100 1001	01H	110 1001	01H*
000 1010	02H	010 1010	02H*	100 1010	0AH	110 1010	02H
000 1011	07H	010 1011	02H	100 1011	03H	110 1011	01H
000 1100	04H	010 1100	0CH	100 1100	04H*	110 1100	04H
000 1101	07H	010 1101	05H	100 1101	04H	110 1101	01H
000 1110	07H	010 1110	02H	100 1110	04H	110 1110	06H
000 1111	07H*	010 1111	07H	100 1111	07H	110 1111	0FH
Received Hamming Sequence	**Data Word**	**Received Hamming Sequence**	**Data Word**	**Received Hamming Sequence**	**Data Word**	**Received Hamming Sequence**	**Data Word**
001 0000	00H	011 0000	08H	101 0000	08H	111 0000	08H*
001 0001	09H	011 0001	0BH	101 0001	0DH	111 0001	08H
001 0010	0EH	011 0010	0BH	101 0010	0AH	111 0010	08H
001 0011	0BH	011 0011	0BH*	101 0011	03H	111 0011	0BH
001 0100	0EH	011 0100	0CH	101 0100	0DH	111 0100	08H
001 0101	0DH	011 0101	05H	101 0101	0DH*	111 0101	0DH
001 0110	0EH*	011 0110	0EH	101 0110	0EH	111 0110	06H
001 0111	0EH	011 0111	0BH	101 0111	0DH	111 0111	0FH
001 1000	09H	011 1000	0BH	101 1000	0AH	111 1000	08H
001 1001	09H*	011 1001	09H	101 1001	09H	111 1001	01H
001 1010	0AH	011 1010	02H	101 1010	0AH*	111 1010	0AH
001 1011	09H	011 1011	0BH	101 1011	0AH	111 1011	0FH
001 1100	0CH	011 1100	0CH*	101 1100	04H	111 1100	0CH
001 1101	09H	011 1101	0CH	101 1101	0DH	111 1101	0FH
001 1110	0EH	011 1110	0CH	101 1110	0AH	111 1110	0FH
001 1111	07H	011 1111	0FH	101 1111	0FH	111 1111	0FH*

Table 2. Decode table for 7/4 Hamming sequences. Asterisks indicate "no errors detected." All other entries have single errors detected.

Implementing the Hamming codes

Why, if they're so easy to implement, aren't Hamming codes more popular? First, there's no agreed-upon protocol. Second, the string of four data bits isn't long enough. The Baudot code uses five bits, with extra characters (FIGS/LTRS) to shift back and forth between two 32-character alphabets. ASCII uses seven bits, and eight are preferred to allow full data transfer. Can't you just break an ASCII word into two four-bit "nibbles," encode and send each, then decode, correct, and add them back up at the receiver?

In theory, you can. But you'll encounter another HF error — fading. Fading causes the entire loss, or "erasure," of one or several words. And, if the receiver was expecting a LSB "nibble" when the fade started, but picked up a most significant bit (MSB) nibble when it ended, it would assemble an incorrect word. Because the receiver's definition of MSB and LSB is now out of sync, all subsequent words will be reassembled incorrectly.

In manual systems like straight RTTY, you could treat this problem the same way you'd treat a FIGS/LTRS garble. You'd use a key to direct the computer to shift the order in which it's reassembling the nibbles.

You could also devote one of the four bits as a flag, indicating whether it's an MSB or LSB nibble. This would leave six bits — enough to encode a 64-character alphabet like Baudot.

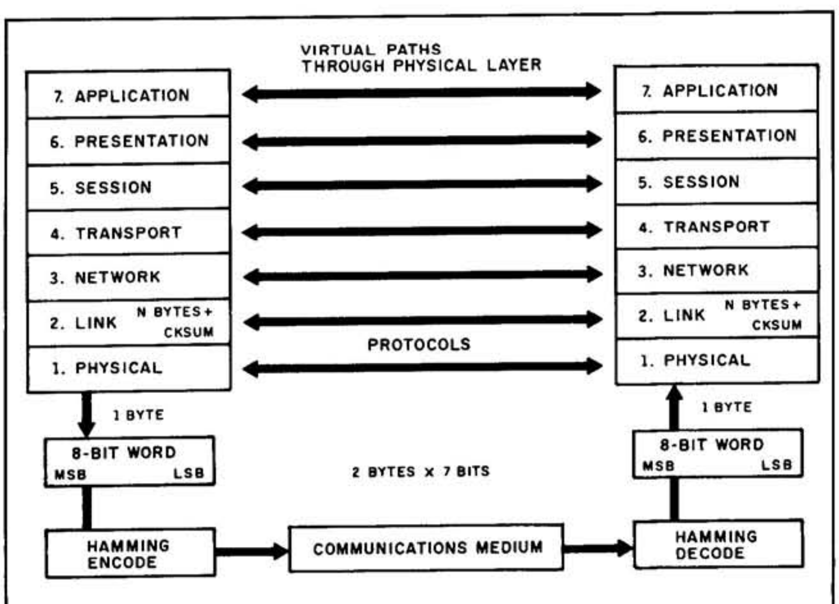

Figure 1. Hamming application to AX.25.

Packet applications

A third possibility would be to place the Hamming codes inside an error-detecting block code. You could send a block of a fixed number of characters (say 127), compute a checksum of each character, and then send the checksum. The receiver would perform a similar process, acknowledging the text if it agrees with the checksum, or asking for a repeat if it's incorrect. This common algorithm for data transfer is used by XMODEM for landline and in the link protocol in AX.25 packet.

The AX.25 packet protocol is organized in "layers." The link layer organizes a block, computes checksums, determines if a block is received correctly, and handles repeat requests. The bottom-most layer is the physical layer. This layer is normally concerned only with modulation and demodulation. It's at the physical layer that you'd intercept a single eight-bit word on its way to the modulator, encode it, and reverse the process at the receiver. **Figure 1** shows how you could include Hamming encoding and decoding at this level without disturbing any of the other layers, except, perhaps, to allow more time for the longer Hamming codes. What do you gain by doing this?

The AX.25 protocol already accounts for packets of incorrect length. Thus, erasures, and the framing problems they may generate, can be detected and handled by requests for retransmission generated by the existing link protocol. And, because the checksum is also encoded as a Hamming sequence, errors which can't be corrected will probably be detected as well, resulting in a retransmission request. You can do all of this without touching the higher-order packet protocols. In all probability, this scheme will correct *all* single-bit errors per Hamming sequence, and detect all erasures (incorrect packet length) and uncorrectable errors (bad checksum). How high is this probability? Let's see.

Number crunching

The Bit Error Rate (BER) is the probability that any bit will be changed. The basic packet word is eight bits long, and each bit must be correct. The probability of receiving an entirely correct eight-bit word is:

$$P(8 \text{ correct bits}) = (1 - BER)^8 \qquad (1)$$

Packets come in various lengths; 128 words is representative. The last word is a checksum, allowing errors in the block to be detected. All 128 words must be received correctly. The probability that 128 correct eight-bit words, or one entire packet of representative length, will be received is:

$$P(\text{correct packet}) = P(8 \text{ correct bits})^{128} \quad \textbf{(2)}$$

The probability that a correct packet will be received is defined as the number of successes divided by the number of attempts. The inverse of this is the average number of attempts that must be made to get one good packet through:

$$N(\text{attempts}) = 1/P(\text{correct packets}) \quad \textbf{(3)}$$

The results are plotted on the graph in **Figure 2**. They show that, without error correction, the number of attempts remains at essentially one transmission per packet up to about 0.0001 BER (1 bit per 10,000 altered). The number of attempts then rises quickly to an average of two transmissions at a BER of 0.0005 and three at 0.001. Beyond that level, the number of attempts required to successfully get a packet through become astronomical.

By comparison, a Hamming sequence will be accepted if it has either no errors, or a single-bit error. Because a Hamming sequence is seven bits long, the no-error probability is:

$$P(7 \text{ correct bits}) = (1 - \text{BER})^7 \quad \textbf{(4)}$$

and the probability of exactly one error is:

$$P(\text{exactly 1 bit error in 7}) = 7{*}\text{BER}{*}(1 - \text{BER})^6 \quad \textbf{(5)}$$

Thus, the probability of no errors, or exactly one error, is the sum of **Equations 4** and **5**:

$$P(0 \text{ or } 1 \text{ error in } 7) = P(7 \text{ correct bits}) + P(\text{exactly 1 bit error in 7}) \quad \textbf{(6)}$$

Because you can only encode four bits onto a seven-bit Hamming sequence, you need 256 Hamming words with one or zero errors each, to convey 128 words with no errors. The probability of this occurring is:

$$P(256 \text{ correctable sequences}) = P(1 \text{ or } 0 \text{ errors in } 7)^{256} \quad \textbf{(7)}$$

And, like the eight-bit word, the number of attempts required is:

$$N(\text{Hamming attempts}) = 1/P(256 \text{ correctable sequences}) \quad \textbf{(8)}$$

This is also plotted in **Figure 2**. You can see that Hamming sequences require no retransmittal until there's a BER of 0.005 – nearly twice the BER that will bring an uncorrected link to its knees! A Hamming encoded link can maintain a useful throughput with less than two attempts per packet until it reaches a BER of 0.02 – nearly 20 times higher than the link without error correction.

Of course, this doesn't come without cost. You may have noticed that Hamming sequences are 7/4 longer; that is, they are nearly twice as long as the unprotected packet. Does this overhead pay for itself?

Yes. The number of bits per packet is the basic number of bits per individual packet times the average number of attempts:

$$\text{TXBITS(Hamming)} = 7{*}256{*}N(\text{Hamming attempts}) \quad \textbf{(9)}$$

and

$$\text{TXBITS(Normal)} = 8{*}128{*}N(\text{attempts}) \quad \textbf{(10)}$$

These are plotted in **Figure 3**. Note that for low error rates, the uncorrected link without Hamming codes outperforms the Hamming link by almost 2:1, requiring only 1024 bits compared with 1792 Hamming bits. The link errors are too few to justify the high overhead of the Hamming bits. At about 0.0005 BER, the two are equal in performance. On the average, the Hamming link will require less than two transmissions for BERs up to 0.01.

This analysis doesn't include the possibility of using the more robust, but slower, Hamming codes to support HF data rates which could go as high as 1200 baud – unthinkably fast for conventional HF packet.

Application to AMTOR

What about AMTOR? AMTOR uses a seven-bit alphabet, with just four 1s and three 0s. A total of 35 characters can be

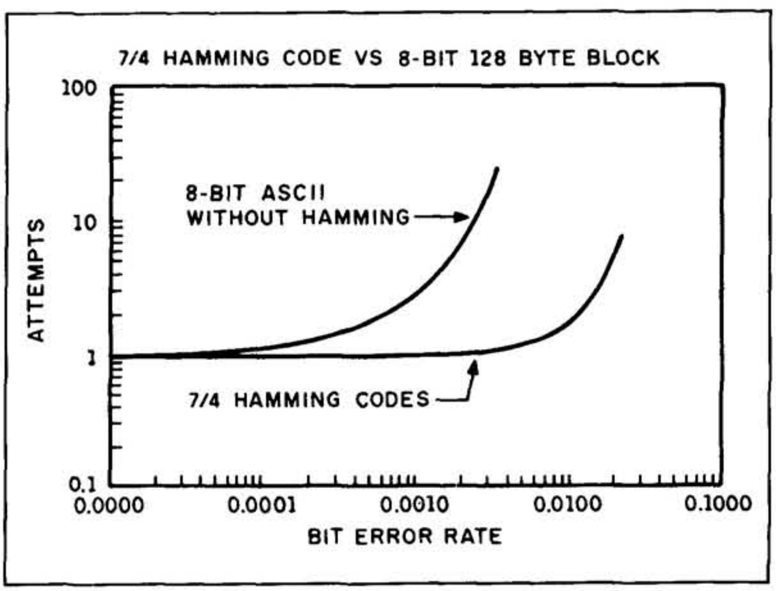

Figure 2. Attempts versus BER, 128-byte packet.

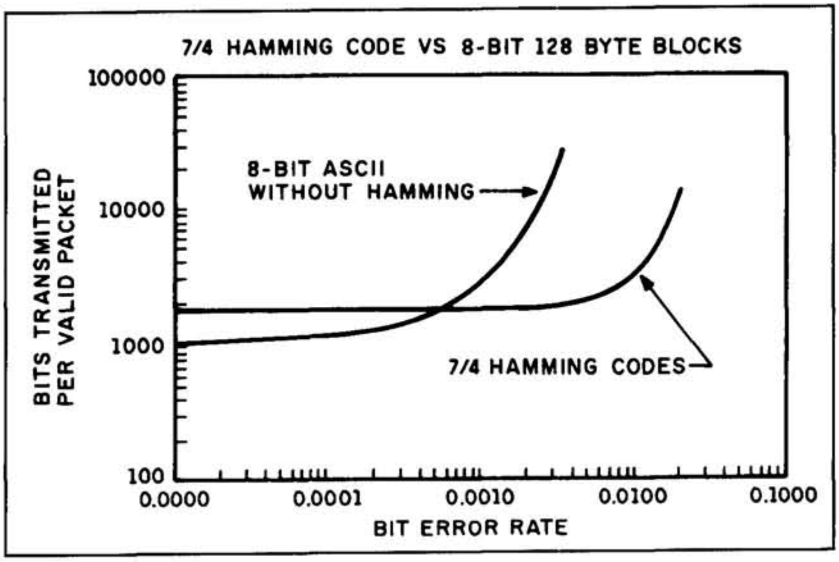

Figure 3. Average transmitted bits per 128-byte packet.

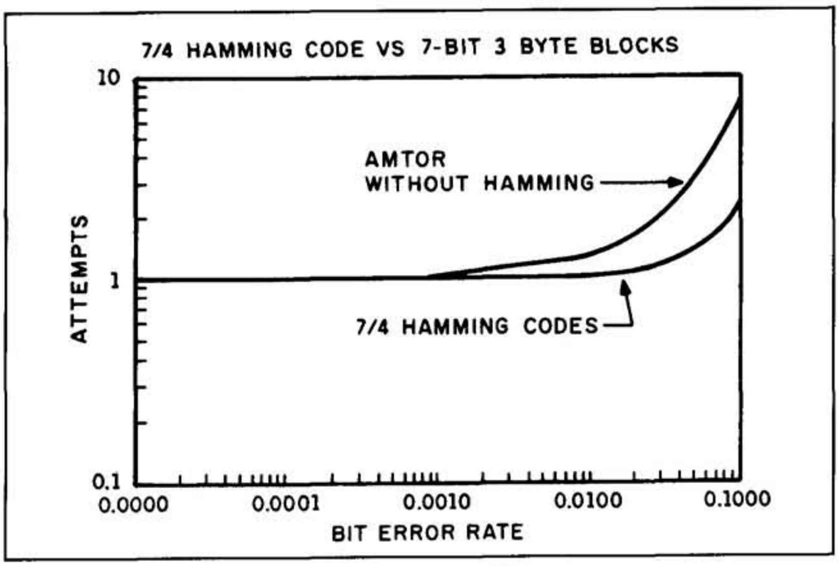

Figure 4. Attempts versus BER, AMTOR 3 bytes × 7 bit group.

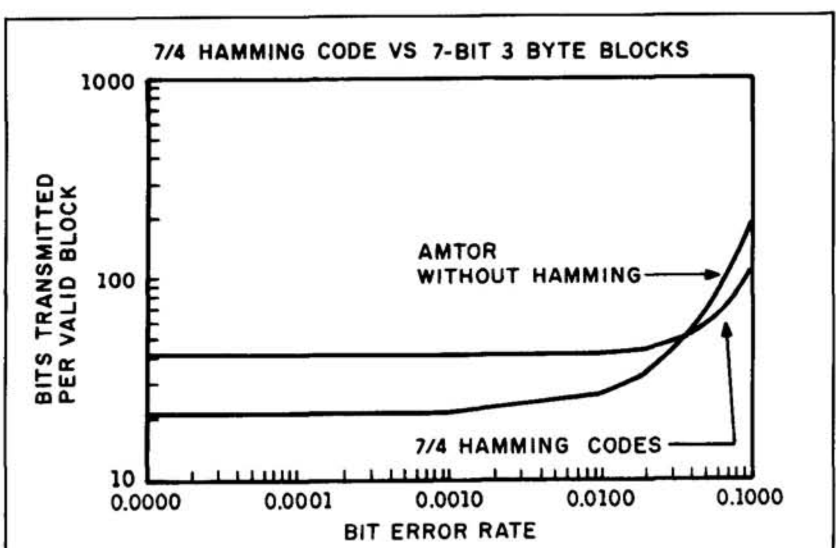

Figure 5. Average transmitted bits per AMTOR 3 bytes × 7 bit group.

encoded. Characters are sent in groups of three. Any character which doesn't confirm to this 4/3 sequence is detected as an error, and generates an RQ (Repeat Request).[3]

The analysis is basically the same as it is for packet. The exception is that the basic block is three seven-byte words, which can be encoded onto six seven-byte Hamming sequences. **Figures 4** and **5** show the results. Surprisingly, Hamming sequences have little advantage over short AMTOR sequences. Uncorrected AMTOR more than holds its own until enormously high BERs of 0.05 are reached. Only in the presence of absolutely incredible noise levels of 0.1, does AMTOR Hamming gain a 2:1 advantage over straight AMTOR. These results shouldn't come as a great surprise to AMTOR enthusiasts. The key lies in the very short AMTOR block lengths. However, I haven't taken the effects of false acknowledgement into consideration here – specifically the misinterpretation of an "ACK" (Acknowledgement) as an "RQ" (Repeat Request).[3]

Frankly, AMTOR doesn't appear to be much improved by Hamming sequences. Using a higher data rate wouldn't change this situation much because AMTOR spends a significant amount of time waiting for transmitters to change over. Shortening the data transmission time wouldn't appear to reduce the overall time significantly. AMTOR efficiency might improve if a longer sequence with Hamming codes is used, but that would involve a major change to the AMTOR protocol.

Technical details of Hamming codes

How do Hamming codes work? To understand how they work, you must first understand the concept of Hamming distance. Hamming distance is the number of bits that would have to be changed to transform one binary word into another. For example: 0011, 0101, 1001, and 0000 are all within one Hamming distance of 0001. By contrast, 1110 is separated by four bits distance from 0001, and all four bits would have to be altered to change 1 (01H) to 14 (0EH). Hamming distance can be computed by "exclusive OR'ing" the two binary words together and counting the 1s.

Hamming sequences which can correct single-bit errors use an "alphabet" in which all the legal sequences that can be transmitted have a Hamming distance of three from each other. For example, **Table 1**, the encode table used in the text, has sixteen seven-bit sequences out of a possible 128, all of which differ from each other by three or more bits. The other 112 possibilities represent erroneous sequences caused by noise that creates a one-bit change in one of the sixteen legal sequences. Because the legal sequences are separated from each other by three bits, these erroneous sequences will be separated by two or more bits from all other legal sequences – except for the one which was actually sent. Using the example in the text, if the sequence encoding 04H is altered at the LSB to 100 1101, this sequence is still at least two bits removed from any other **Table 1** sequence, except 100 1100. Thus, you can assume that an erroneous sequence should actually be the legal sequence "closest" to it.

Two-bit errors still won't produce a legal sequence. They won't decode correctly, either. Hamming codes of distance three can't distinguish between a correctable single-bit error and an uncorrectable two-bit error.

Generating the Hamming sequence

Hamming sequences use parity bits based on the message word to be encoded. The parity bits are defined so they will actually point to zero (the error-free condition), or a binary number representing the bit position in error. Because of this, bits in a Hamming sequence are numbered from 1 to N, rather than the binary starting point of zero. Three parity bits are needed to handle a seven-bit sequence, leaving four bits available as message bits.

Table 3 shows how parity bits are defined for 7/4, 15/11 (four parity bits), and 31/26 (five parity bits) sequences. Parity bits are assigned to locations corresponding to integer powers of two within the sequence (bits P1, P2, P4, P8, and P16), and message bits to all others. A "1" at the intersection of a message bit row and parity bit column, means that the message bit should be included when determining the corresponding parity bit. Following the example in the text, "1s" appear opposite M3, M5, and M7, under parity bit P1. These bits are XOR'd together to form the parity bit 1. P2 is the XOR of message bits M3, M6, and M7, and P4 is the XOR of M5, M6, and M7.

Hamming sequences were originally intended for hardware generation and detection.[4] **Figure 6A** shows how the 7/4 code above can be created in hardware for the transmitter. The parity bits are woven into the sequence as shown. This figure illustrates the generation of the 1001100 sequence for 04H, given in the text.

The receiver in **Figure 6B** generates the same parity bits – P1, P2, and P4 – and XOR's each with its corresponding received parity. The resulting three-bit word is called the "syndrome," and points to the binary position of the bit in error. As shown, the receiver copies 1001101, generating a syndrome of 7. The syndrome is applied to a 1-of-8 decoder. Zero is the "no errors detected" condition; 1, 2, and 4 indicate that the parity bits themselves were in error and aren't needed to correct the message bits. Finally, 3, 5, 6, and 7 are XOR'd with their corresponding message bits. Bit 7 in the example is XOR'd by the decoded syndrome, back to its correct value of 0.

The Hamming codes were developed when such hardware solutions were essential for implementing the technique. Solutions of this type are still necessary for longer sequences where the decode tables can become prohibitively large. However, as noted in the text, look-up table schemes in software are now a more efficient implementation for short sequences like the 7/4 code.[2]

	P16	P8	P4	P2	P1
M3	0	0	0	1	1
M5	0	0	1	0	1
M6	0	0	1	1	0
M7	0	0	1	1	1
			(7/4 sequences)		
M9	0	1	0	0	1
M10	0	1	0	1	0
M11	0	1	0	1	1
M12	0	1	1	0	0
M13	0	1	1	0	1
M14	0	1	1	1	0
M15	0	1	1	1	1
		(15/11 sequences)			
M17	1	0	0	0	1
M18	1	0	0	1	0
M19	1	0	0	1	1
M20	1	0	1	0	0
M21	1	0	1	0	1
M22	1	0	1	1	0
M23	1	0	1	1	1
M24	1	1	0	0	0
M25	1	1	0	0	1
M26	1	1	0	1	0
M27	1	1	0	1	1
M28	1	1	1	0	0
M29	1	1	1	0	1
M30	1	1	1	1	0
M31	1	1	1	1	1
	(31/26 sequences)				

Table 3. General scheme for determining parity for 7/4, 15/11, and 31/26 Hamming sequences. A "1" indicates that the corresponding message bit should be included in the XOR tree for that particular parity bit.

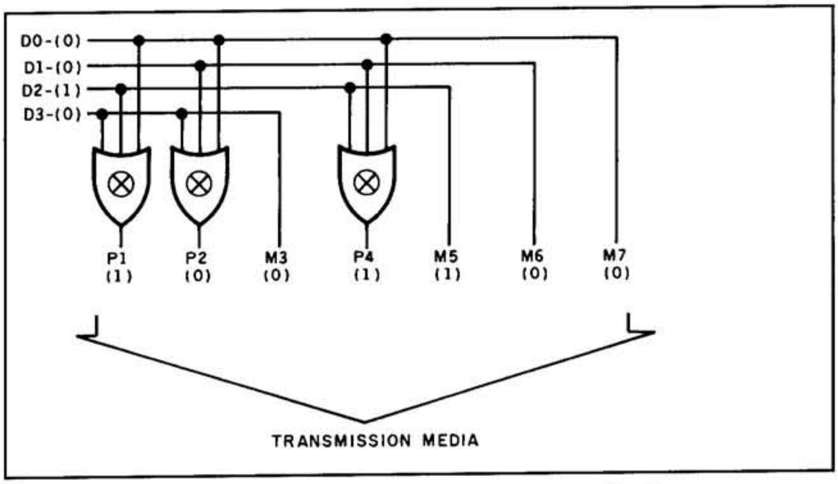

Figure 6A. Hardware encoding of data 0100 onto 7/4 sequence 1001100.

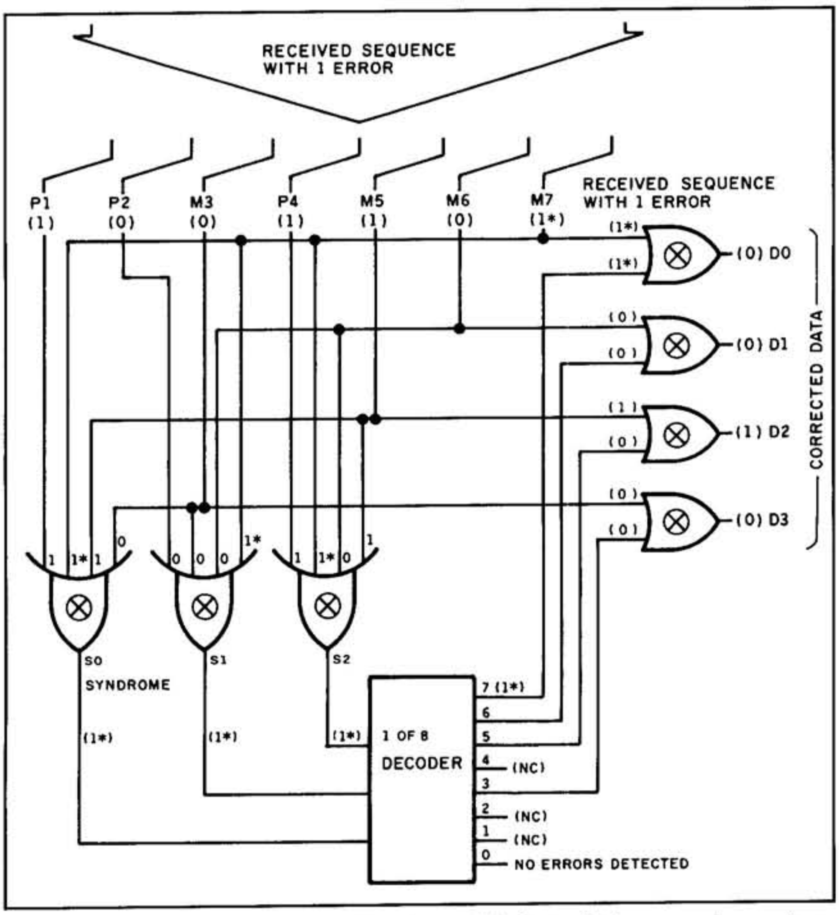

Figure 6B. Hardware decoding of a 7/4 receiver. "*" indicates bit in error and corrections.

Hamming codes assume that two-bit errors within a word are much less likely to appear than single-bit errors. This isn't strictly true. Many errors occur in bursts, causing multiple errors and even complete loss, or erasure, of entire the word. Nevertheless, Hamming codes can easily correct most errors in communications.

Summary

The main limitation to the more widespread implementation of this simple scheme is the lack of an accepted protocol. The techniques I've described here are easily implemented on any computer/TU system capable of straight ASCII operation. The integration of this technique into the AX.25 packet protocol may be a bit more complicated, but it would improve HF packet throughput significantly. In fact, Hamming code applications could improve this form of packet to such a dramatic extent, that some serious research may be in order.

I invite all who wish to experiment with the development of a new protocol based on the Hamming technique to contact me, either by mail or at my packet address, KB6IC @ KØBOY. ■

References

1. R.W. Hamming, "Error Decting and Error Correcting Codes," *Bell System Technical Journal*, Volume 29, April 1950, pages 147-160.
2. Ben White, "Hamming-Code Decoding," *Dr. Dobbs Journal*, October 1989, pages 52-56.
3. *The ARRL 1986 Handbook*, 63rd Edition, The American Radio Relay League, Newington, Connecticut, 1986, page 19-10.
4. John D. Lenk, *Handbook of Logic Circuits*, Reston Publishing Company, Reston, Virginia, 1972, pages 90-93.