CHAPTER 15

ERROR CORRECTING CODES

Insert this material after Chapter 13 (Gray Code) and after the new material on the Cyclic Redundancy Check.

15–1 Introduction

This section is a brief introduction to the theory and practice of error correcting codes (ECCs). We limit our attention to binary forward error correcting (FEC) block codes. This means that the symbol alphabet consists of just two symbols (which we denote 0 and 1), that the receiver can correct a transmission error without asking the sender for more information or for a retransmission, and that the transmissions consist of a sequence of fixed length blocks, called *code words.*

Section 15–2 describes the code independently discovered by R. W. Hamming and M. J. E. Golay before 1950 [Ham]. This code is single error correcting (SEC), and a simple extension of it, also discovered by Hamming, is single error correcting and, simultaneously, double error detecting (SEC-DED).

Section 15–4 steps back and asks what is possible in the area of forward error correction. Still sticking to binary FEC block codes, the basic question addressed is: for a given block length (or *code length*) and level of error detection and correction capability, how many different code words can be encoded?

Section 15–2 is for readers who are primarily interested in learning the basics of how ECC works in computer memories. Section 15–4 is for those who are interested in the mathematics of the subject, and who might be interested in the challenge of an unsolved mathematical problem.

The reader is cautioned that over the past 50 years ECC has become a very big subject. Many books have been published on it and closely related subjects [Hill, LC, MS, and Roman, to mention only a few]. Here we just scratch the surface and introduce the reader to two important topics and to some of the terminology used in this field. Although much of the subject of error correcting codes relies very heavily on the notations and results of linear algebra, and in fact is a very nice application of that abstract theory, we avoid it here for the benefit of those who are not familiar with that theory.

The following notation is used throughout this chapter. It is close to that used in [LC]. The terms are defined in subsequent sections.

k	Number of "information" or "message" bits.
m	Number of parity-check bits ("check bits," for short).
n	Code length, $n = m + k$.

$\boldsymbol{u}$ Information bit vector, $\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots \boldsymbol{u}_{k-1}$.
$\boldsymbol{p}$ Parity check bit vector, $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_{m-1}$.
$\boldsymbol{s}$ Syndrome vector, $\boldsymbol{s}_0, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_{m-1}$.

15–2 The Hamming Code

Hamming's development [Ham] is a very direct construction of a code that permits correcting single-bit errors. He assumes that the data to be transmitted consists of a certain number of *information bits* $\boldsymbol{u}$, and he adds to these a number of *check bits* $\boldsymbol{p}$ such that if a block is received that has at most one bit in error, then $\boldsymbol{p}$ identifies the bit that is in error (which may be one of the check bits). Specifically, in Hamming's code $\boldsymbol{p}$ is interpreted as an integer which is 0 if no error occurred, and otherwise is the 1-origined index of the bit that is in error. Let k be the number of information bits, and m the number of check bits used. Because the m check bits must check themselves as well as the information bits, the value of $\boldsymbol{p}$, interpreted as an integer, must range from 0 to $m + k$, which is $m + k + 1$ distinct values. Because m bits can distinguish 2^m cases, we must have

$$2^m \geq m + k + 1. \tag{1}$$

This is known as the *Hamming rule*. It applies to any single error correcting (SEC) binary FEC block code in which all of the transmitted bits must be checked. The check bits will be interspersed among the information bits in a manner described below.

Because $\boldsymbol{p}$ indexes the bit (if any) that is in error, the least significant bit of $\boldsymbol{p}$ must be 1 if the erroneous bit is in an odd position, and 0 if it is in an even position or if there is no error. A simple way to achieve this is to let the least significant bit of $\boldsymbol{p}$, $\boldsymbol{p}_0$, be an even parity check on the odd positions of the block, and to put $\boldsymbol{p}_0$ in an odd position. The receiver then checks the parity of the odd positions (including that of $\boldsymbol{p}_0$). If the result is 1, an error has occurred in an odd position, and if the result is 0, either no error occurred or an error occurred in an even position. This satisfies the condition that $\boldsymbol{p}$ should be the index of the erroneous bit, or be 0 if no error occurred.

Similarly, let the next from least significant bit of $\boldsymbol{p}$, $\boldsymbol{p}_1$, be an even parity check of positions 2, 3, 6, 7, 10, 11, … (in binary, 10, 11, 110, 111, 1010, 1011, …), and put $\boldsymbol{p}_1$ in one of these positions. Those positions have a 1 in their second from least significant binary position number. The receiver checks the parity of these positions (including the position of $\boldsymbol{p}_1$). If the result is 1, an error occurred in one of those positions, and if the result is 0, either no error occurred or an error occurred in some other position.

Continuing, the third from least significant check bit, $\boldsymbol{p}_2$, is made an even parity check on those positions that have a 1 in their third from least significant position number, namely positions 4, 5, 6, 7, 12, 13, 14, 15, 20, …, and $\boldsymbol{p}_2$ is put in one of those positions.

Putting the check bits in power-of-two positions (1, 2, 4, 8, …) has the advantage that they are independent. That is, the sender can compute $\boldsymbol{p}_0$ independently of $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, … and, more generally, it can compute each check bit independently of the others.

As an example, let us develop a single error correcting code for $k = 4$. Solving (1) for m gives $m = 3$, with equality holding. This means that all 2^m possible values of the m check bits are used, so it is particularly efficient. A code with this property is called a *perfect* code.[1]

This code is called the (7,4) Hamming code, which signifies that the code length is 7 and the number of information bits is 4. The positions of the check bits $\boldsymbol{p}_i$ and the information bits $\boldsymbol{u}_i$ are shown below.

$\boldsymbol{p}_0$	$\boldsymbol{p}_1$	$\boldsymbol{u}_3$	$\boldsymbol{p}_2$	$\boldsymbol{u}_2$	$\boldsymbol{u}_1$	$\boldsymbol{u}_0$
1	2	3	4	5	6	7

Table 15–1 shows the entire code. The 16 rows show all 16 possible information bit configurations and the check bits calculated by Hamming's method.

TABLE 15–1. THE (7,4) HAMMING CODE

Information	**1** $\boldsymbol{p}_0$	**2** $\boldsymbol{p}_1$	**3** $\boldsymbol{u}_3$	**4** $\boldsymbol{p}_2$	**5** $\boldsymbol{u}_2$	**6** $\boldsymbol{u}_1$	**7** $\boldsymbol{u}_0$
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1
10	1	0	1	1	0	1	0
11	0	1	1	0	0	1	1
12	0	1	1	1	1	0	0
13	1	0	1	0	1	0	1
14	0	0	1	0	1	1	0
15	1	1	1	1	1	1	1

1. A perfect code exists for $k = 2^m - m - 1$, m an integer—that is, $k = 1, 4, 11, 26, 57, 120, \ldots$.

To illustrate how the receiver corrects a single-bit error, suppose the code word

1001110

is received. This is row 4 in Table 15–1 with bit 6 flipped. The receiver calculates the *exclusive or* of the bits in odd positions and gets 0. It calculates the *exclusive or* of bits 2, 3, 6, and 7 and gets 1. Lastly it calculates the *exclusive or* of bits 4, 5, 6, and 7 and gets 1. Thus the error indicator, which is called the *syndrome*, is binary 110, or 6. The receiver flips the bit at position 6 to correct the block.

A SEC-DED Code

For many applications a single error correcting code would be considered unsatisfactory, because it accepts all blocks received. A SEC-DED code seems safer, and it is the level of correction and detection most often used in computer memories.

The Hamming code can be converted to a SEC-DED code by adding one check bit, which is a parity bit (let us assume even parity) on all the bits in the SEC code word. This code is called an *extended Hamming code* [Hill, MS]. It is not obvious that it is SEC-DED. To see that it is, consider Table 15–2. It is assumed *a*

TABLE 15–2. ADDING A PARITY BIT TO MAKE A SEC-DED CODE

Possibilities			Receiver Conclusion
Errors	Overall Parity	Syndrome	
0	even	0	No error
1	odd	0	Overall parity bit is in error
		≠0	Syndrome indicates the bit in error
2	even	≠0	Double error (not correctable)

priori that either 0, 1, or 2 transmission errors occur. As indicated in the table, if there are no errors, the overall parity (the parity of the entire *n*-bit received code word) will be even, and the syndrome of the $(n-1)$-bit SEC portion of the block will be 0. If there is one error, then the overall parity of the received block will be odd. If the error occurred in the overall parity bit, then the syndrome will be 0. If the error occurred in some other bit, then the syndrome will be nonzero and it will indicate which bit is in error. If there are two errors, then the overall parity of the received block will be even. If one of the two errors is in the overall parity bit, then the other is in the SEC portion of the block. In this case the syndrome will be nonzero (and will indicate the bit in the SEC portion that is in error). If the errors are both in the SEC portion of the block, then the syndrome will also be nonzero, although the reason is a bit hard to explain.

The reason is that there must be a check bit that checks one of the two bit positions but not the other one. The parity of this check bit and the bits it checks will thus be odd, resulting in a nonzero syndrome. Why must there be a check bit

that checks one of the erroneous bits but not the other one? To see this, first suppose one of the erroneous bits is in an even position and the other is in an odd position. Then because one of the check bits ($\boldsymbol{p}_0$) checks all the odd positions and none of the even positions, the parity of the bits at the odd positions will be odd, resulting in a nonzero syndrome. More generally, suppose the erroneous bits are in positions i and j (with $i \neq j$). Then because the binary representations of i and j must differ in some bit position, one of them has a 1 at that position and the other has a 0 at that position. The check bit corresponding to this position in the binary integers checks the bits at positions in the code word that have a 1 in their position number, but not the positions that have a 0 in their position number. The bits covered by that check bit will have odd parity, and thus the syndrome will be nonzero. As an example, suppose the erroneous bits are in positions 3 and 7. In binary, the position numbers are 0…0011 and 0…0111. These numbers differ in the third position from the right, and at that position, the number 7 has a 1 and the number 3 has a 0. Therefore the bits checked by the third check bit (these are bits 4, 5, 6, 7, 12, 13, 14, 15, …) will have odd parity.

Thus, referring to Table 15–2, the overall parity and the syndrome together uniquely identify whether 0, 1, or 2 errors occurred. In the case of one error, the receiver can correct it. In the case of two errors, the receiver cannot tell whether just one of the errors is in the SEC portion (in which case it could correct it) or both errors are in the SEC portion (in which case an attempt to correct it would result in incorrect information bits).

The overall parity bit could as well be a parity check on only the even positions, because the overall parity bit is easily calculated from that and the parity of the odd positions (which is the least significant check bit). More generally, the overall parity bit could as well be a parity check on the complement set of bits checked by any one of the SEC parity bits. This observation might save some gates in hardware.

It should be clear that the Hamming SEC code is of minimum redundancy. That is, for a given number of information bits, it adds a minimum number of check bits that permit single error correction. This is true because it was constructed that way. Hamming shows that the SEC-DED code constructed from a SEC code by adding one overall parity bit is also of minimum redundancy. His argument is to assume that a SEC-DED code exists that has fewer check bits, and he derives from this a contradiction to the fact that the starting SEC code had minimum redundancy.

Minimum Number of Check Bits Required

The middle column of Table 15–3 shows minimal solutions of inequality (1) for a range of values of k. The rightmost column simply shows that one more bit is required for a SEC-DED code. From this table one can see, for example, that to provide the SEC-DED level ECC for a memory word containing 64 information bits, eight check bits are required, giving a total memory word size of 72 bits.

TABLE 15–3. EXTRA BITS FOR ERROR CORRECTION/DETECTION

Number of Information Bits k	m for SEC	m for SEC-DED
1	2	3
2 to 4	3	4
5 to 11	4	5
12 to 26	5	6
27 to 57	6	7
58 to 120	7	8
121 to 247	8	9
248 to 502	9	10

Concluding Remarks

In the more mathematically oriented ECC literature, the term "Hamming code" is reserved for the perfect codes described above—that is, those with $(n, k) = (3, 1)$, $(7, 4)$, $(15, 11)$, $(31, 26)$, and so on. Similarly, the extended Hamming codes are the perfect SEC-DED codes described above. However, computer architects and engineers often use the term to denote any of the codes that Hamming described, and some variations. The term "extended" is often understood.

The first IBM computer to use Hamming codes was the IBM Stretch computer (model 7030), built in 1961 [LC]. It used a (72, 64) SEC-DED code (not a perfect code). A follow-on machine known as Harvest (model 7950), built in 1962, was equipped with 22-track tape drives that employed a (22, 16) SEC-DED code. The ECCs found on modern machines are usually not Hamming codes, but rather are codes devised for some logical or electrical property such as minimizing the depth of the parity check trees, and making them all the same length. Such codes give up Hamming's simple method of determining which bit is in error, and instead use a hardware table lookup.

At the time of this writing (2005), most notebook PCs (personal computers) have no error checking in their memory systems. Desktop PCs may have none, or they may have a simple parity check. Server-class computers generally have ECC at the SEC-DED level.

In the early solid state computers equipped with ECC memory, the memory was usually in the form of eight check bits and 64 information bits. A memory module (group of chips) might be built from, typically, nine eight-bit wide chips. A word access (72 bits, including check bits) fetches eight bits from each of these nine chips. Each chip is laid out in such a way that the eight bits accessed for a single word are physically far apart. Thus, a word access references 72 bits that are physically somewhat separated. With bits interleaved in that way, if a few close-together bits in the same chip are altered, as for example by an alpha particle or cosmic ray hit, a few *words* will have *single-bit* errors, which can be corrected.

Some larger memories incorporate a technology known as *Chipkill*. This allows the computer to continue to function even if an entire memory chip fails, for example due to loss of power to the chip.

The interleaving technique can be used in communication applications to correct burst errors, by interleaving the bits in time.

Today the organization of ECC memories is often more complicated than simply having eight check bits and 64 information bits. Modern server memories may have 16 or 32 information bytes (128 or 256 bits) checked as a single ECC word. Each DRAM chip may store two, three, or four bits in physically adjacent positions. Correspondingly, ECC is done on alphabets of four, eight, or 16 characters—a subject not discussed here. Because the DRAM chips usually come in 8- or 16-bit wide configurations, the memory module often provides more than enough bits for the ECC function. The extra bits might be used for other functions, such as one or two parity bits on the memory address. This allows the memory to check that the address it receives is (probably) the address that the CPU generated.

In modern server-class machines, ECC may be used in different levels of cache memory, as well as in main memory. It may also be used in non-memory areas, such as on busses.

15–3 Software for SEC-DED on 32 Information Bits

This section describes a code for which encoding and decoding can be efficiently implemented in software for a basic RISC. It does single error correction and double error detection on 32 information bits. The technique is basically Hamming's.

We follow Hamming in using check bits in such a way that the receiver can easily (in software) determine whether zero, one, or two errors occurred, and if one error occurred it can easily correct it. We also follow Hamming in using a single overall parity bit to convert a SEC code to SEC-DED, and we assume the check bit values are chosen to make even parity on the check bit and the bits it checks. A total of seven check bits are required (Table 15–3).

Consider first just the SEC property, without DED. For SEC, six check bits are required. For implementation in software, the main difficulty with Hamming's method is that it merges the six check bits with the 32 information bits, resulting in a 38-bit quantity. We are assuming the implementation is done on a 32-bit machine, and the information bits are in a 32-bit word. It would be very awkward for the sender to spread out the information bits over a 38-bit quantity and calculate the check bits into the positions described by Hamming. The receiver would have similar difficulties. The check bits could be moved into a separate word or register, with the 32 information bits kept in another word or register. But this gives an irregular range of positions that are checked by each check bit. In the scheme to be described, these ranges retain most of the regularity that they have in Hamming's scheme (which ignores word boundaries). The regularity leads to simplified calculations.

The positions checked by each check bit are shown in Table 15–4. In this table, bits are numbered in the usual little-endian way, with position 0 being the least significant bit (unlike Hamming's numbering).

TABLE 15–4. POSITIONS CHECKED BY THE CHECK BITS

Check Bit	Positions Checked
$\boldsymbol{p}_0$	0, 1, 3, 5, 7, 9, 11, …, 29, 31
$\boldsymbol{p}_1$	0, 2–3, 6–7, 10–11, …, 30–31
$\boldsymbol{p}_2$	0, 4–7, 12–15, 20–23, 28–31
$\boldsymbol{p}_3$	0, 8–15, 24–31
$\boldsymbol{p}_4$	0, 16–31
$\boldsymbol{p}_5$	1–31

Observe that each of the 32 information word bit positions is checked by at least two check bits. For example, position 6 is checked by $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$. Thus if two information words differ in one bit position, the code words (information plus check bits) differ in three positions, so they are at a distance of at least three from one another (see "Hamming Distance" on page 13). Furthermore, if two information words differ in two bit positions, then at least one of $\boldsymbol{p}_0$–$\boldsymbol{p}_5$ checks one of the positions but not the other, so again the code words will be at least a distance three apart. Therefore, the above scheme represents a code with minimum distance three (a SEC code).

Suppose a code word is transmitted to a receiver. Let $\boldsymbol{u}$ denote the information bits received, $\boldsymbol{p}$ denote the check bits received, and $\boldsymbol{s}$ (for syndrome) denote the *exclusive or* of $\boldsymbol{p}$ and the check bits calculated from $\boldsymbol{u}$ by the receiver. Then examination of Table 15–4 reveals that $\boldsymbol{s}$ will be set as shown in Table 15–5, for zero or one errors in the code word.

TABLE 15–5. SYNDROME FOR ZERO OR ONE ERRORS

Error in bit	Resulting syndrome $\boldsymbol{s}_5 \ldots \boldsymbol{s}_0$
(no errors)	000000
$\boldsymbol{u}_0$	011111
$\boldsymbol{u}_1$	100001
$\boldsymbol{u}_2$	100010
$\boldsymbol{u}_3$	100011
$\boldsymbol{u}_4$	100100
…	…
$\boldsymbol{u}_{30}$	111110
$\boldsymbol{u}_{31}$	111111

TABLE 15–5. SYNDROME FOR ZERO OR ONE ERRORS

Error in bit	Resulting syndrome $s_5 \ldots s_0$
p_0	000001
p_1	000010
p_2	000100
p_3	001000
p_4	010000
p_5	100000

As an example, suppose information bit u_4 is corrupted in transmission. Table 15–4 shows that u_4 is checked by check bits p_2 and p_5. Therefore the check bits calculated by the sender and receiver will differ in p_2 and p_5. In this scenario the check bits received are the same as those transmitted, so the syndrome will have bits 2 and 5 set—that is, it will be 100100.

If one of the check bits is corrupted in transmission (and no errors occur in the information bits), then the check bits received and those calculated by the receiver (which equal those calculated by the sender) differ in the check bit that was corrupted, and in no other bits, as shown in the last six rows of Table 15–5.

The syndromes shown in Table 15–5 are distinct, for all 39 possibilities of no error or a single-bit error anywhere in the code word. Therefore the syndrome identifies whether or not an error occurred, and if so, which bit position is in error. Furthermore, if a single-bit error occurred, it is fairly easy to calculate which bit is in error (without resorting to a table lookup), and to correct it. The logic is:

If $s = 0$, no error occurred.
If $s = 011111$, u_0 is in error.
If $s = 1xxxxx$, with xxxxx nonzero, the error is in u at position xxxxx.
Otherwise, a single bit in s is set, the error is in a check bit, and the correct check bits are given by the *exclusive or* of the syndrome and the received check bits (or by the calculated check bits).

Under the assumption that an error in the check bits need not be corrected, this may be expressed as shown below, where b is the bit number to be corrected.

```
if (s & (s – 1)) = 0 then …       // No correction required.
else do
  if s = 0b011111 then b ← 0
  else b ← s & 0b011111
  u ← u ⊕ (1 ≪ b)                 // Complement bit b of u.
end
```

The code of Figure 15–2 uses a hack that changes the second if-then-else construction shown above into an assignment statement.

To recognize double-bit errors, an overall parity bit is computed (parity of $\boldsymbol{u}_{31:0}$ and $\boldsymbol{p}_{5:0}$), and put in bit position 6 of $\boldsymbol{p}$ for transmission. Double-bit errors are distinguished by the overall parity being correct, but with the syndrome ($\boldsymbol{s}_{5:0}$) being nonzero. The reason the syndrome is nonzero is the same as in the case of the extended Hamming code, given on page 4.

Software that implements this code is shown in Figures 15–1 and 15–2. We assume the simple case of a sender and a receiver, and the receiver has no need to correct an error that occurs in the check bits or in the overall parity bit.

To compute the check bits, function `checkbits` first ignores information bit $\boldsymbol{u}_0$ and computes

0. $(\boldsymbol{x} = \boldsymbol{u} \oplus (\boldsymbol{u} \overset{u}{\gg} 1))$
1. $(\boldsymbol{x} = \boldsymbol{x} \oplus (\boldsymbol{x} \overset{u}{\gg} 2))$
2. $(\boldsymbol{x} = \boldsymbol{x} \oplus (\boldsymbol{x} \overset{u}{\gg} 4))$
3. $(\boldsymbol{x} = \boldsymbol{x} \oplus (\boldsymbol{x} \overset{u}{\gg} 8))$
4. $(\boldsymbol{x} = \boldsymbol{x} \oplus (\boldsymbol{x} \overset{u}{\gg} 16))$

except omitting line i when computing check bit $\boldsymbol{m}_i$, for $0 \le i \le 4$. For $\boldsymbol{p}_5$, all the above assignments are used. This is where the regularity of the pattern of bits checked by each check bit pays off; a lot of code commoning can be done. This reduces what would be $4 \times 5 + 5 = 25$ such assignments to 15, as shown in Figure 15–1.

Incidentally, if the computer has an instruction for computing the parity of a word, or has the *population count* instruction (which puts the word parity in the least significant bit of the target register), then the regular pattern is not needed. On such a machine, the check bits might be computed as

```
p0 = pop(u ^ 0xAAAAAAAB) & 1;
p1 = pop(u & 0xCCCCCCCD) & 1;
```

and so forth.

After packing the six check bits into a single quantity `p`, the `checkbits` function accounts for information bit $\mathtt{u}_0$ by complementing all six check bits if $\mathtt{u}_0 = 1$. (C.f. Table 15–4; $\mathtt{p}_5$ must be complemented because $\mathtt{u}_0$ was erroneously included in the calculation of $\mathtt{p}_5$ up to this point.)

.

```
unsigned int checkbits(unsigned int u) {

   /* Computes the six parity check bits for the
   "information" bits given in the 32-bit word u. The
   check bits are p[5:0]. On sending, an overall parity
   bit will be prepended to p (by another process).

   Bit   Checks these bits of u
   p[0]  0, 1, 3, 5, ..., 31 (0 and the odd positions).
   p[1]  0, 2-3, 6-7, ..., 30-31 (0 and positions xxx1x).
   p[2]  0, 4-7, 12-15, 20-23, 28-31 (0 and posns xx1xx).
   p[3]  0, 8-15, 24-31 (0 and positions x1xxx).
   p[4]  0, 16-31 (0 and positions 1xxxx).
   p[5]  1-31 */

   unsigned int p0, p1, p2, p3, p4, p5, p6, p;
   unsigned int t1, t2, t3;

   // First calculate p[5:0] ignoring u[0].
   p0 = u ^ (u >> 2);
   p0 = p0 ^ (p0 >> 4);
   p0 = p0 ^ (p0 >> 8);
   p0 = p0 ^ (p0 >> 16);        // p0 is in posn 1.

   t1 = u ^ (u >> 1);
   p1 = t1 ^ (t1 >> 4);
   p1 = p1 ^ (p1 >> 8);
   p1 = p1 ^ (p1 >> 16);        // p1 is in posn 2.

   t2 = t1 ^ (t1 >> 2);
   p2 = t2 ^ (t2 >> 8);
   p2 = p2 ^ (p2 >> 16);        // p2 is in posn 4.

   t3 = t2 ^ (t2 >> 4);
   p3 = t3 ^ (t3 >> 16);        // p3 is in posn 8.

   p4 = t3 ^ (t3 >> 8);         // p4 is in posn 16.

   p5 = p4 ^ (p4 >> 16);        // p5 is in posn 0.

   p = ((p0>>1) & 1) | ((p1>>1) & 2) | ((p2>>2) & 4) |
       ((p3>>5) & 8) | ((p4>>12) & 16) | ((p5 & 1) << 5);

   p = p ^ (-(u & 1) & 0x3F);   // Now account for u[0].
   return p;
}
```

FIGURE 15–1. Calculation of check bits.

```
int correct(unsigned int pr, unsigned int *ur) {

   /* This function looks at the received seven check
   bits and 32 information bits (pr and ur), and
   determines how many errors occurred (under the
   presumption that it must be 0, 1, or 2). It returns
   with 0, 1, or 2, meaning that no errors, one error, or
   two errors occurred. It corrects the information word
   received (ur) if there was one error in it. */

   unsigned int po, p, syn, b;

   po = parity(pr ^ *ur);       // Compute overall parity
                                // of the received data.
   p = checkbits(*ur);          // Calculate check bits
                                // for the received info.
   syn = p ^ (pr & 0x3F);       // Syndrome (exclusive of
                                // overall parity bit).
   if (po == 0) {
      if (syn == 0) return 0;   // If no errors, return 0.
      else return 2;            // Two errors, return 2.
   }
                                // One error occurred.
   if (((syn - 1) & syn) == 0)  // If syn has zero or one
      return 1;                 // bits set, then the
                                // error is in the check
                                // bits or the overall
                                // parity bit (no
                                // correction required).

   // One error, and syn bits 5:0 tell where it is in ur.

   b = syn - 31 - (syn >> 5); // Map syn to range 0 to 31.
// if (syn == 0x1f) b = 0;    // (These two lines equiv.
// else b = syn & 0x1f;       // to the one line above.)
   *ur = *ur ^ (1 << b);      // Correct the bit.
   return 1;
}
```

FIGURE 15–2. The receiver's actions.

15–4 Error Correction Considered More Generally

This section continues to deal with only the binary FEC block codes, but a little more generally than the codes described in Section 15–2. We drop the assumption that the block consists of a set of "information" bits and a distinct set of "check" bits, and any implication that the number of code words must be a power of 2. We also consider levels of error correction and detection capability greater than SEC and SEC-DED. For example, suppose you want a double error correcting code for decimal digits. If the code has 16 code words (with ten being used to represent the decimal digits and six being unused), the length of the code words must be at least 11. But if a code with only 10 code words is used, the code words may be of length 10. (This is shown in Table 15–8 on page 21, in the column for $d = 5$, as is explained below.)

A *code* is simply a set of *code words*, and for our purposes the code words are binary strings all of the same length which, as mentioned above, is called the *code length*. The number of code words in the set is called the *code size*. We make no interpretation of the code words; they may represent alphanumeric characters or pixel values in a picture, for example.

As a trivial example, a code might consist of the binary integers from 0 to 7, with each bit repeated three times:

{000000000, 000000111, 000111000, 000111111, 111000000, … 111111111}.

Another example is the *two-out-of-five* code, in which each code word has exactly two 1-bits:

{00011, 00101, 00110, 01001, 01010, 01100, 10001, 10010, 10100, 11000}.

The code size is 10, and thus it is suitable for representing decimal digits. Notice that if codeword 00110 is considered to represent decimal 0, then the remaining values can be decoded into digits 1 through 9 by giving the bits weights of 6, 3, 2, 1, and 0, in left-to-right order.

The *code rate* is a measure of the efficiency of a code. For a code like Hamming's, this can be defined as the number of information bits divided by the code length. For the Hamming code discussed above it is $4/7 \approx 0.57$. More generally, the code rate is defined as the log base 2 of the code size divided by the code length. The simple codes above have rates of $\log_2(8)/9 \approx 0.33$ and $\log_2(10)/5 \approx 0.66$, respectively.

Hamming Distance

The central concept in the theory of ECC is that of *Hamming distance*. The Hamming distance between two words (of equal length) is the number of bit positions in which they differ. Put another way, it is the population count of the *exclusive or* of the two words. It is appropriate to call this a distance function because it satisfies the definition of a distance function used in linear algebra:

$$d(x, y) = d(y, x),$$
$$d(x, y) \geq 0,$$
$$d(x, y) = 0 \quad \text{iff} \quad x = y, \quad \text{and}$$
$$d(x, y) + d(y, z) \geq d(x, z) \quad \text{(triangle inequality)}.$$

Here $d(x, y)$ denotes the Hamming distance between code words x and y, which for brevity we will call simply the *distance* between x and y.

Suppose a code has a minimum distance of 1. That is, there are two words x and y in the set that differ in only one bit. Clearly, if x were transmitted and the bit that makes it distinct from y were flipped due to a transmission error, then the receiver could not distinguish between receiving x with a certain bit in error, and receiving y with no errors. Hence in such a code it is impossible to detect even a 1-bit error, in general.

Suppose now that a code has a minimum distance of 2. Then if just one bit is flipped in transmission, an invalid code word is produced, and thus the receiver can (in principle) detect the error. But if two bits are flipped, a valid code word might be transformed into another valid code word. Thus double-bit errors cannot be detected. Furthermore, single-bit errors cannot be *corrected*. This is because if a received word has one bit in error, then there may be two code words that are one bit-change away from the received word, and the receiver has no basis for deciding which is the original code word.

The code obtained by appending a single parity bit is in this category. It is shown below for the case of three information bits ($k = 3$). The rightmost bit is the parity bit, chosen to make even parity on all four bits. The reader may verify that the minimum distance between code words is 2.

0000
0011
0101
0110
1001
1010
1100
1111

Actually, adding a single parity bit permits detecting any odd number of errors, but when we say that a code permits detecting m-bit errors, we mean *all* errors up to m bits.

Now consider the case in which the minimum distance between code words is 3. If any one or two bits is flipped in transmission, an invalid code word results. If just one bit is flipped, the receiver can (we imagine) try flipping each of the received bits one at a time, and in only one case will a code word result. Hence in such a code the receiver can detect and correct a single-bit error. However, a double-bit error might appear to be a single-bit error from another code word, and thus the receiver cannot detect double-bit errors.

Similarly, it is easy to reason that if the minimum distance of a code is 4, the receiver can correct all single-bit errors and detect all double-bit errors (it is a SEC-DED code). As mentioned above, this is the level of capability often used in computer memories.

Table 15–6 summarizes the error correction and detection capabilities of a block code based on its minimum distance.

TABLE 15–6. NUMBER OF BITS CORRECTED/DETECTED

Minimum Distance	Correct	Detect
1	0	0
2	0	1
3	1	1
4	1	2
5	2	2
6	2	3
7	3	3
8	3	4
d	$\lfloor (d-1)/2 \rfloor$	$\lfloor d/2 \rfloor$

Error correction capability can be traded for error detection. For example, if the minimum distance of a code is 3, that redundancy can be used to correct no errors but to detect single- or double-bit errors. If the minimum distance is 5, the code can be used to correct single-bit errors and detect 3-bit errors, or to correct no errors but to detect 4-bit errors, and so forth. Whatever is subtracted from the "Correct" column of Table 15–6 can be added to the "Detect" column.

The Main Coding Theory Problem

Up to this point we have asked, "Given a number of information bits k and a desired minimum distance d, how many check bits are required?" In the interest of generality, we will now turn this question around and ask "For a given code length n and minimum distance d, how many code words are possible?" Thus, the number of code words need not be an integral power of 2.

Following [Roman] and others, let $A(n, d)$ denote the largest possible code size for a (binary) code with length n and minimum distance d. The remainder of this section is devoted to exploring some of what is known about this function. Determining its values has been called *the main coding theory problem* [Hill, Roman]. Throughout this section we assume that $n \geq d \geq 1$.

It is nearly trivial that

$$A(n, 1) = 2^n, \tag{2}$$

because there are 2^n distinct words of length n.

For minimum distance 2, we know from the single parity bit example that $A(n, 2) \geq 2^{n-1}$. But $A(n, 2)$ cannot exceed 2^{n-1} for the following reason. Suppose there is a code of length n and minimum distance 2 that has more than 2^{n-1} code words. Delete any one column from the code words. (We envision the code words as being arranged in a matrix much like that of Table 15–1.) This produces a code of length $n-1$ and minimum distance at least 1 (deleting a column can reduce the minimum distance by at most 1), and of size exceeding 2^{n-1}. Thus it has $A(n-1, 1) > 2^{n-1}$, contradicting equation (2). Hence

$$A(n, 2) = 2^{n-1}.$$

That was not difficult. What about $A(n, 3)$? That is an unsolved problem, in the sense that no formula or reasonably easy means of calculating it is known. Of course many specific values of $A(n, 3)$ are known, and some bounds are known, but the exact value is unknown in most cases.

When equality holds in (1), it represents the solution to this problem for the case $d = 3$. Letting $n = m + k$, (1) may be rewritten

$$2^k \leq \frac{2^n}{n+1}. \tag{3}$$

Here k is the number of information bits, so 2^k is the number of code words. Hence we have

$$A(n, 3) \leq \frac{2^n}{n+1},$$

with equality holding when $2^n/(n+1)$ is an integer.

For $n = 7$, this gives $A(n, 3) \leq 16$, and we know from Hamming's construction that 16 can be achieved. For $n = 3$ it gives $A(n, 3) \leq 2$, which can be realized with code words 000 and 111. For $n = 4$ it gives $A(n, 3) \leq 3.2$, and with a little doodling you will see that it is not possible to get three code words with $n = 4$ and $d = 3$. Thus when equality does not hold in (3), it merely gives an upper bound, quite possibly not realizable, on the maximum number of code words.

An interesting relation is that for $n \geq 2$,

$$A(n, d) \leq 2A(n-1, d). \tag{4}$$

Thus adding 1 to the code length at most doubles the number of code words possible, for the same minimum distance d. To see this, suppose you have a code of length n, distance d, and size $A(n, d)$. Choose an arbitrary column of the code. Either half or more of the code words have a 0 in the selected column, or half or more have a 1 in that position. Of these two subsets, choose one that has at least $A(n, d)/2$ code words, form a new code consisting of this subset, and delete the

selected column (which is either all 0's or all 1's). The resulting set of code words has n reduced by 1, has the same distance d, and has at least $A(n, d)/2$ code words. Thus $A(n-1, d) \geq A(n, d)/2$, from which inequality (4) follows.

A useful relation is that if d is even, then

$$A(n, d) = A(n-1, d-1). \tag{5}$$

To see this, suppose you have a code C of length n and minimum distance d, with d odd. Form a new code by appending to each word of C a parity bit, let us say to make the parity of each word even. The new code has length $n+1$, and has the same number of code words as does C. It has minimum distance $d+1$. For if two words of C are a distance x apart, with x odd, then one word must have even parity and the other must have odd parity. Thus we append a 0 in the first case and a 1 in the second case, which increases the distance between the words to $x+1$. If x is even, we append a 0 to both words, which does not change the distance between them. Because d is odd, all pairs of words that are a distance d apart become distance $d+1$ apart. The distance between two words more than d apart either does not change or increases. Therefore the new code has minimum distance $d+1$. This shows that if d is odd, then $A(n+1, d+1) \geq A(n, d)$, or equivalently $A(n, d) \geq A(n-1, d-1)$ for even $d \geq 2$.

Now suppose you have a code of length n and minimum distance $d \geq 2$ (d can be odd or even). Form a new code by eliminating any one column. The new code has length $n-1$, minimum distance at least $d-1$, and is the same size as the original code (all the code words of the new code are distinct because the new code has minimum distance at least 1). Therefore $A(n-1, d-1) \geq A(n, d)$. This establishes equation (5).

Spheres

Upper and lower bounds on $A(n, d)$, for any $d \geq 1$, can be derived by thinking in terms of n-dimensional spheres. Given a code word, think of it as being at the center of a "sphere" of radius r, consisting of all words at a Hamming distance r or less from it.

How many points (words) are in a sphere of radius r? First consider how many points are in the shell at distance exactly r from the central code word. This is given by the number of ways to choose r different items from n, ignoring the order of choice. We imagine the r chosen bits as being complemented, to form a word at distance exactly r from the central point. This "choice" function, often written $\binom{n}{r}$, may be calculated from[2]

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}.$$

2. It is also called the "binomial coefficient" because $\binom{n}{r}$ is the coefficient of the term $x^r y^{n-r}$ in the expansion of the binomial $(x+y)^n$.

Thus $\binom{n}{0} = 1$, $\binom{n}{1} = n$, $\binom{n}{2} = \frac{n(n-1)}{2}$, $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$, and so forth.

The total number of points in a sphere of radius r is the sum of the points in the shells from radius 0 to r:

$$\sum_{i=0}^{r} \binom{n}{i}.$$

There seems to be no simple formula for this sum [Knu1].

From this it is easy to obtain bounds on $A(n, d)$. First, assume you have a code of length n and minimum distance d, and it consists of M code words. Surround each code word with a sphere, all of the same maximal radius such that no two spheres have a point in common. This radius is $(d-1)/2$ if d is odd, and is $(d-2)/2$ if d is even (see Figure 15–3). Because each point is in at most one sphere, the total number of points in the M spheres must be less than or equal to the total number of points in the space. That is,

$$M \sum_{i=0}^{\lfloor (d-1)/2 \rfloor} \binom{n}{i} \leq 2^n.$$

This holds for any M, hence for $M = A(n, d)$, so that

$$A(n, d) \leq \frac{2^n}{\sum_{}^{\lfloor (d-1)/2 \rfloor} \binom{n}{i}}.$$

This is known as the *sphere-packing bound*, or the *Hamming bound*.

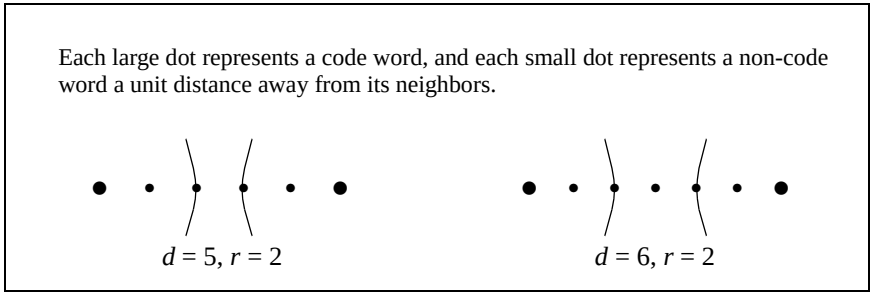

FIGURE 15–3. Maximum radius that allows correcting points within a sphere.

The sphere idea also easily gives a lower bound on $A(n, d)$. Assume again that you have a code of length n and minimum distance d, and it has the maximum possible number of code words—that is, it has $A(n, d)$ code words. Surround each code word with a sphere of radius $d-1$. Then these spheres must cover *all* 2^n points in the space (possibly overlapping). For if not, there would be a point that is at a distance d or more from all code words, and that is impossible because such

a point would be a code word. Thus we have a weak form of the *Gilbert-Varshamov* bound:

$$A(n, d) \sum_{i=0}^{d-1} \binom{n}{i} \geq 2^n.$$

There is the strong form of the G-V bound, which applies to *linear* codes. Its derivation relies on methods of linear algebra which, important as they are to the subject of linear codes, are not covered in this short introduction to error correcting codes. Suffice it to say that a linear code is one in which the sum (*exclusive or*) of any two code words is also a code word. The Hamming code of Table 15–1 is a linear code. Because the G-V bound is a lower bound on linear codes, it is also a lower bound on the unrestricted codes considered here. For large n, it is the best known lower bound on both linear and unrestricted codes.

The strong G-V bound states that $A(n, d) \geq 2^k$, where k is the largest integer such that

$$2^k < \frac{2^n}{\sum_{i=0}^{d-2} \binom{n-1}{i}}.$$

That is, it is the value of the right-hand side of this inequality rounded down to the next strictly smaller integral power of 2. The "strictness" is important for cases such as $(n, d) = (8, 3)$, $(16, 3)$ and (the degenerate case) $(6, 7)$.

Combining these results:

$$\mathrm{GP2LT}\left(\frac{2^n}{\sum_{i=0}^{d-2} \binom{n-1}{i}}\right) \leq A(n, d) \leq \frac{2^n}{\sum_{i=0}^{\lfloor (d-1)/2 \rfloor} \binom{n}{i}}, \tag{6}$$

where GP2LT denotes the greatest integral power of 2 (strictly) less than its argument.

Table 15–7 gives the values of these bounds for some small values of n and d. A single number in an entry means the lower and upper bounds given by (6) are equal.

If d is even, bounds can be computed directly from (6) or, making use of equation (5), they can be computed from (6) with d replaced with $d-1$ and n replaced with $n-1$ in the two bounds expressions. It turns out that the latter method always results in tighter or equal bounds. Therefore, the entries in Table 15–7 were calculated only for odd d. To access the table for even d, use the values of d shown in the footing and the values of n shown at the right (shaded regions).

The bounds given by (6) can be seen to be rather loose, especially for large d. The ratio of the upper bound to the lower bound diverges to infinity with increasing n. The lower bound is particularly loose. Over a thousand papers have been

TABLE 15–7. THE G-V AND HAMMING BOUNDS ON $A(n, d)$

n	$d = 4$	$d = 6$	$d = 8$	$d = 10$	$d = 12$	$d = 14$	$d = 16$	n
6	4 – 5	2	–	–	–	–	–	**5**
7	8 – 9	2	–	–	–	–	–	**6**
10	32 – 51	4 – 11	2 – 3	2	–	–	–	**9**
13	256 – 315	16 – 51	2 – 13	2 – 5	2	–	–	**12**
16	2048	64 – 270	8 – 56	2 – 16	2 – 6	2 – 3	2	**15**
19	8192 – 13797	256 – 1524	16 – 265	4 – 64	2 – 20	2 – 8	2 – 4	**18**
22	65536 – 95325	1024 – 9039	64 – 1342	8 – 277	4 – 75	2 – 25	2 – 10	**21**
25	2^{19} – 671088	4096 – 55738	256 – 7216	32 – 1295	8 – 302	2 – 88	2 – 31	**24**
28	2^{22} – 793490	32768 – 354136	1024 – 40622	128 – 6436	16 – 1321	4 – 337	2 – 104	**27**
	$d = 3$	**$d = 5$**	**$d = 7$**	**$d = 9$**	**$d = 11$**	**$d = 13$**	**$d = 15$**	

written describing methods to improve these bounds, and the results as of this writing are shown in Table 15–8 [Agrell, Brou; where they differ, the table shows the tighter bounds].

The cases of $(n, d) = (7, 3)$, $(15, 3)$, and $(23, 7)$ are *perfect* codes, meaning that they achieve the upper bound given by (6). This definition is a generalization of that given on page 3. The codes for which n is odd and $n = d$ are also perfect; see exercise 6.

We conclude this section by pointing out that the idea of minimum distance over an entire code, which leads to the ideas of p-bit error detection and q-bit error correction for some p and q, is not the only criterion for the "power" of a binary FEC block code. For example, work has been done on codes aimed at correcting burst errors. [Etzion] has demonstrated a (16, 11) code, and others, that can correct any single-bit error and any error in two consecutive bits, and is perfect, in a sense not discussed here. It is not capable of general double-bit error detection. The (16, 11) extended Hamming code is SEC-DED and is perfect. Thus his code gives up general double-bit error *detection* in return for double-bit error *correction* of consecutive bits. This is of course interesting because in many applications errors are likely to occur in short bursts.

TABLE 15–8. BEST KNOWN BOUNDS ON $A(n, d)$

n	$d = 4$	$d = 6$	$d = 8$	$d = 10$	$d = 12$	$d = 14$	$d = 16$	n
6	4	2	–	–	–	–	–	**5**
7	8	2	–	–	–	–	–	**6**
8	16	2	2	–	–	–	–	**7**
9	20	4	2	–	–	–	–	**8**
10	40	6	2	2	–	–	–	**9**
11	72	12	2	2	–	–	–	**10**
12	144	24	4	2	2	–	–	**11**
13	256	32	4	2	2	–	–	**12**
14	512	64	8	2	2	2	–	**13**
15	1024	128	16	4	2	2	–	**14**
16	2048	256	32	4	2	2	2	**15**
17	2720 – 3276	256 – 340	36 – 37	6	2	2	2	**16**
18	5312 – 6552	512 – 680	64 – 72	10	4	2	2	**17**
19	10496 – 13104	1024 – 1280	128 – 142	20	4	2	2	**18**
20	20480 – 26208	2048 – 2372	256 – 274	40	6	2	2	**19**
21	36864 – 43688	2560 – 4096	512	42 – 48	8	4	2	**20**
22	73728 – 87376	4096 – 6941	1024	64 – 87	12	4	2	**21**
23	147456 – 173015	8192 – 13766	2048	80 – 150	24	4	2	**22**
24	294912 – 344308	16384 – 24106	4096	128 – 280	48	6	4	**23**
25	2^{19} – 599184	16384 – 48008	4096 – 5477	192 – 503	52 – 56	8	4	**24**
26	2^{20} – 1198368	32768 – 84260	4096 – 9672	384 – 859	64 – 98	14	4	**25**
27	2^{21} – 2396736	65536 – 157285	8192 – 17768	512 – 1764	128 – 169	28	6	**26**
28	2^{22} – 4793472	131072 – 291269	16384 – 32151	1024 – 3200	178 – 288	56	8	**27**
	$d = 3$	**$d = 5$**	**$d = 7$**	**$d = 9$**	**$d = 11$**	**$d = 13$**	**$d = 15$**	

Exercises

1. Show a Hamming code for $k = 3$ (make a table similar to Table 15–1).

2. In a certain application of a SEC code, there is no need to correct the check bits. Thus the m check bits need only check the information bits, but not themselves. For k information bits, m must be large enough so that the receiver can distinguish $k + 1$ cases: which of the k bits is in error, or no error occurred. Thus the number of check bits required is given by $2^m \geq k + 1$. This is a weaker restriction on m than is the Hamming rule, so it should be possible to construct, for some values of k, a SEC code that has fewer check bits than those required by the Hamming rule. Alternatively, one could have just one value to signify that an error occurred somewhere in the check bits, without specifying where. This would lead to the rule $2^m \geq k + 2$.

What is wrong with this reasoning?

3. Prove: $A(2n, 2d) \geq A(n, d)$.

4. Prove the "singleton bound": $A(n, d) \leq 2^{n-d+1}$.

5. Show that the notion of a perfect code as equality in the right-hand portion of inequality (6) is a generalization of the Hamming rule.

6. What is the value of $A(n, d)$ if $n = d$? Show that for odd n, these codes are perfect.

7. Show that if n is a multiple of 3 and $d = 2n/3$, then $A(n, d) = 4$.

8. Show that if $d > 2n/3$, $A(n, d) = 2$.

9. (Brain teaser) How would you find, numerically, the minimum m that satisfies (1), as a function of k?

References

[Agrell] Agrell, Erik. http://www.s2.chalmers.se/~agrell/bounds/, October 2003.

[Brou] Brouwer, Andries E. http://www.win.tue.nl/~aeb/binary-1.html, March 2004.

[Etzion] Etzion, Tuvi. "Constructions for Perfect 2-Burst-Correcting Codes," *IEEE Transactions on Information Theory 47*, 6 (September 2001), 2553–2555.

[Ham] Hamming, Richard W., "Error Detecting and Error Correcting Codes," *The Bell System Technical Journal 26*, 2 (April 1950), 147–160. NB: We interchange the roles of the variables *k* and *m* relative to how they are used by Hamming. We follow the more modern usage found in [LC] and [MS], for example.

[Hill] Hill, Raymond. *A First Course in Coding Theory.* Clarendon Press, 1986.

[LC] Lin, Shu and Costello, Daniel J., Jr. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.

[MS] MacWilliams, Florence J. and Sloane, Neil. J. A. *The Theory of Error-Correcting Codes Part II*. North-Holland, 1977.

[Roman] Roman, Steven. *Coding and Information Theory.* Springer-Verlag, 1992.

Answers to Exercises

1. Your table should look like Table 15–1 with the rightmost column and the odd numbered rows deleted.

2. In the first case, if an error occurs in a check bit, the receiver cannot know that, and it will make an erroneous "correction" to the information bits.

In the second case, if an error occurs in a check bit, the syndrome will be one of 100…0, 010…0, 001…0, …, 000…1 (m distinct values). Therefore m must be large enough to encode these m values, as well as the k values to encode a single error in one of the k information bits, and a value for "no errors." So the Hamming rule stands.

One thing along these lines that could be done is to have a single parity bit for the m check bits, and have the m check bits encode values of one error in an information bit (and where it is), or no errors occurred. For this code, m could be chosen as the smallest value for which $2^m \geq k + 1$. The code length would be $k + m + 1$, where the "+1" is for the parity bit on the check bits. But this code length is nowhere better than that given by the Hamming rule, and is sometimes worse.

3. Given a code of length n and minimum distance d, simply double-up each 1 and each 0 in each code word. The resulting code is of length $2n$, minimum distance $2d$, and is the same size.

4. Given a code of length n, minimum distance d, and size $A(n, d)$, think of it as being displayed as in Table 15–1. Remove an arbitrary $d - 1$ columns. The resulting code words, of length $n - (d - 1)$, have minimum distance at least 1. That is, they are all distinct. Hence their number cannot be more than $2^{n-(d-1)}$. Since deleting columns did not change the code size, the original code's size is at most $2^{n-(d-1)}$, so that $A(n, d) \leq 2^{n-d+1}$.

5. The Hamming rule applies to the case that $d = 3$ and the code has 2^k code words, where k is the number of information bits. The right-hand part of inequality (6), with $A(n, d) = 2^k$ and $d = 3$ is

$$2^k \leq \frac{2^n}{\binom{n}{0} + \binom{n}{1}} = \frac{2^n}{1 + n}.$$

Replacing n with $m + k$ gives

$$2^k \leq \frac{2^{m+k}}{1 + m + k},$$

which on cancelling 2^k on each side becomes inequality (1).

6. The code must consist of an arbitrary bit string and its one's-complement, so its size is 2. That these codes are perfect, for odd n, can be seen by

showing that they achieve the upper bound in inequality (6). Proof sketch: An n-bit binary integer may be thought of as representing uniquely a choice from n objects, with a 1-bit meaning to choose and a 0-bit meaning not to choose the corresponding object. Therefore there are 2^n ways to choose from 0 to n objects from n objects—that is, $\sum_{i=0}^{n} \binom{n}{i} = 2^n$. If n is odd, i ranging from 0 to $(n-1)/2$ covers half the terms of this sum, and because of the symmetry $\binom{n}{i} = \binom{n}{n-i}$, it accounts for half the sum. Therefore $\sum_{i=0}^{(n-1)/2} \binom{n}{i} = 2^{n-1}$, so that the upper bound in (6) is 2. Thus the code achieves the upper bound of (6).

7. For ease of exposition, this proof will make use of the notion of *equivalence* of codes. Clearly a code is not changed in any substantial way by rearranging its columns (as depicted in Figure 15–1), or by complementing any column. If one code can be derived from another by such transformations, they are called *equivalent*. Because a code is an unordered set of code words, the order of a display of its code words is immaterial. By complementing columns, any code can be transformed into an equivalent code that has a code word that is all 0's.

Also for ease of exposition, we carry out this proof for the case $n = 9$ and $d = 6$.

Wlog (without loss of generality), let code word 0 (the first, which we will call cw_0) be 000 000 000. Then all other code words must have at least six 1's, to differ from cw_0 in at least six places.

Assume (which will be shown) that the code has at least three code words. Then no code word can have seven or more 1's. For if one did, then another code word (which necessarily has six or more 1's) would have at least four of its 1's in the same columns as the word with seven or more 1's. This means the code words would be equal in four or more positions, and so could differ in five or fewer positions (9 – 4), violating the requirement that $d = 6$. Thus all code words other than the first must have exactly six 1's.

Wlog, rearrange the columns so that the first two code words are:

cw_0: 000 000 000
cw_1: 111 111 000

The next code word, cw_2, cannot have four or more of its 1's in the left six columns, because then it would be the same as cw_1 in four or more positions, and so would differ from cw_1 in five or fewer positions. Therefore it has three or fewer of its 1's in the left six columns, so that three of its 1's must be in the right three positions. Therefore exactly three of its 1's are in the left six columns. Rearrange the left six columns (of all three code words) so that cw_2 looks like this:

cw_2: 111 000 111

By similar reasoning, the next code word (cw_3) cannot have four of its 1's in the left three and right three positions, because it would then equal cw_2 in four positions. Therefore it has three of fewer 1's in the left three and right three positions, so that three of its 1's must be in the middle three positions. By similarly comparing it to cw_1, we conclude that three of its 1's must be in the right three positions. Therefore cw_3 is:

cw_3: 000 111 111

By comparing the next code word, if one is possible, with cw_1, cw_2, and cw_3, we conclude that it must have three 1's in the right three positions, in the middle three positions, and in the left three positions, respectively. This is impossible. By inspection, the above four code words satisfy $d = 6$, so $A(9, 6) = 4$.

8. Obviously $A(n, d)$ is at least 2, because the two code words can be all 0's and all 1's. Reasoning similarly as in the previous exercise, let one code word, cw_0, be all 0's. Then all other code words must have more than $2n/3$ 1's. If the code has three or more code words, then any two code words other than cw_0 must have 1's in the same positions for more than $2n/3 - n/3 = n/3$ positions, as suggested by the figure below.

$$\begin{array}{c} 1111\ldots11110\ldots0 \\ > 2n/3 \quad < n/3 \end{array}$$

(The figure represents cw_1 with its 1's pushed to the left. Imagine placing the more than $2n/3$ 1's of cw_2 to minimize the overlap of the 1's.) Since cw_1 and cw_2 overlap in more than $n/3$ positions, they can differ in less than $n - n/3 = 2n/3$ positions, resulting in a minimum distance less than $2n/3$.

9. Treating m and k as real numbers, the following iteration converges from below quite rapidly:

$$\begin{aligned} m_0 &= 0, \\ m_{i+1} &= \lg(m_i + k + 1), \quad i = 0, 1, \ldots, \end{aligned}$$

where $\lg(x)$ is the log base 2 of x. The correct result is given by $\text{ceil}(m_2)$—that is, only two iterations are required for all $k \geq 0$.

Taking another tack, it is not difficult to prove that for $k \geq 0$,

$$\text{bitsize}(k) \leq m \leq \text{bitsize}(k) + 1.$$

Here $\text{bitsize}(k)$ is the size of k in bits, for example $\text{bitsize}(3) = 2$, $\text{bitsize}(4) = 3$, and so forth. (This is different from the function of the same name described in section 5–3 of *Hacker's Delight*, which is for signed integers.) Hint: $\text{bitsize}(k) = \lceil \lg(k + 1) \rceil = \lfloor \lg(k) + 1 \rfloor$, where we take $\lg(0)$ to be -1. Thus one can try $m = \text{bitsize}(k)$, test it, and if it proves to be too small then simply add 1 to the trial value. Using the *number of leading zeros* function to compute $\text{bitsize}(k)$, one way to commit this to code is:

$$\boldsymbol{m} \leftarrow \boldsymbol{W} - \mathrm{nlz}(\boldsymbol{k}),$$
$$\boldsymbol{m} \leftarrow \boldsymbol{m} + ((1 \ll \boldsymbol{m}) - 1 - \boldsymbol{m} \overset{u}{<} \boldsymbol{k}),$$

where W is the machine's word size and $0 \le k \le 2^W - 1$.