# Error Correction and Detection Handout

John F. Wakerly, Digital Design: Principle and Practices (4th edition) ( Prentice Hall )

Chapter 2-15

## *2.15 Codes for Detecting and Correcting Errors

*error*
*failure*
*temporary failure*
*permanent failure*

An *error* in a digital system is the corruption of data from its correct value to some other value. An error is caused by a physical *failure*. Failures can be either temporary or permanent. For example, a cosmic ray or alpha particle can cause a temporary failure of a memory circuit, changing the value of a bit stored in it. Letting a circuit get too hot or zapping it with static electricity can cause a permanent failure, so that it never works correctly again.

*error model*
*independent error model*
*single error*
*multiple error*

The effects of failures on data are predicted by *error models*. The simplest error model, which we consider here, is called the *independent error model*. In this model, a single physical failure is assumed to affect only a single bit of data; the corrupted data is said to contain a *single error*. Multiple failures may cause *multiple errors*—two or more bits in error—but multiple errors are normally assumed to be less likely than single errors.

### 2.15.1 Error-Detecting Codes

*error-detecting code*
*noncode word*

Recall from our definitions in Section 2.10 that a code that uses *n*-bit strings need not contain $2^n$ valid code words; this is certainly the case for the codes that we now consider. An *error-detecting code* has the property that corrupting or garbling a code word will likely produce a bit string that is *not* a code word (a *noncode word*).

A system that uses an error-detecting code generates, transmits, and stores only code words. Thus, errors in a bit string can be detected by a simple rule—if the bit string is a code word, it is assumed to be correct; if it is a noncode word, it contains an error.

An *n*-bit code and its error-detecting properties under the independent error model are easily explained in terms of an *n*-cube. A code is simply a subset

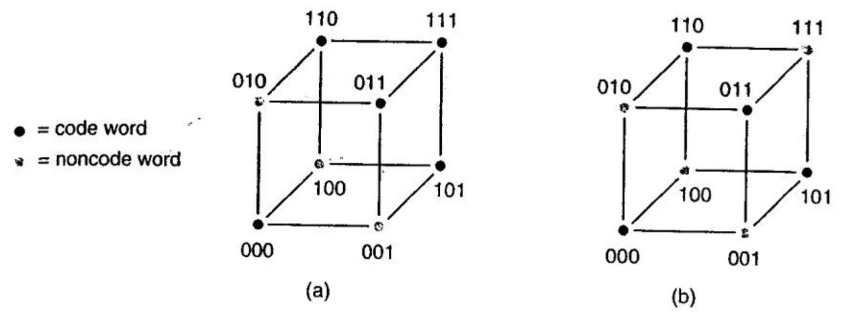

**Figure 2-10** Code words in two different 3-bit codes: (a) minimum distance = 1, does not detect all single errors; (b) minimum distance = 2, detects all single errors.

of the vertices of the *n*-cube. In order for the code to detect all single errors, no code-word vertex can be immediately adjacent to another code-word vertex.

For example, Figure 2-10(a) shows a 3-bit code with five code words. Code word 111 is immediately adjacent to code words 110, 011, and 101. Since a single failure could change 111 to 110, 011, or 101, this code does not detect all single errors. If we make 111 a noncode word, we obtain a code that does have the single-error-detecting property, as shown in (b). No single error can change one code word into another.

The ability of a code to detect single errors can be stated in terms of the concept of distance introduced in the preceding section:

- A code detects all single errors if the *minimum distance* between all possible pairs of code words is 2. *minimum distance*

*information bit*
*parity bit*
*even-parity code*

In general, we need $n + 1$ bits to construct a single-error-detecting code with $2^n$ code words. The first *n* bits of a code word, called *information bits*, may be any of the $2^n$ *n*-bit strings. To obtain a minimum-distance-2 code, we add one more bit, called a *parity bit*, that is set to 0 if there are an even number of 1s among the information bits, and to 1 otherwise. This is illustrated in the first two columns of Table 2-13 for a code with three information bits. A valid $(n+1)$-bit code word has an even number of 1s, and this code is called an *even-parity code*.

| Information Bits | Even-parity Code | Odd-parity Code |
|---|---|---|
| 000 | 000 0 | 000 1 |
| 001 | 001 1 | 001 0 |
| 010 | 010 1 | 010 0 |
| 011 | 011 0 | 011 1 |
| 100 | 100 1 | 100 0 |
| 101 | 101 0 | 101 1 |
| 110 | 110 0 | 110 1 |
| 111 | 111 1 | 111 0 |

**Table 2-13** Distance-2 codes with three information bits.

*odd-parity code*
*1-bit parity code*

We can also construct a code in which the total number of 1s in a valid $(n + 1)$-bit code word is odd; this is called an *odd-parity code* and is shown in the third column of the table. These codes are also sometimes called *1-bit parity codes*, since they each use a single parity bit.

The 1-bit parity codes do not detect 2-bit errors, since changing two bits does not affect the parity. However, the codes can detect errors in any *odd* number of bits. For example, if three bits in a code word are changed, then the resulting word has the wrong parity and is a noncode word. This doesn't help us much, though. Under the independent error model, 3-bit errors are much less likely than 2-bit errors, which are not detectable. Thus, practically speaking, the 1-bit parity codes' error-detection capability stops after 1-bit errors. Other codes, with minimum distance greater than 2, can be used to detect multiple errors.

### 2.15.2 Error-Correcting and Multiple-Error-Detecting Codes

*check bits*

By using more than one parity bit, or *check bits*, according to some well-chosen rules, we can create a code whose minimum distance is greater than 2. Before showing how this can be done, let's look at how such a code can be used to correct single errors or detect multiple errors.

Suppose that a code has a minimum distance of 3. Figure 2-11 shows a fragment of the $n$-cube for such a code. As shown, there are at least two noncode words between each pair of code words. Now suppose we transmit code words

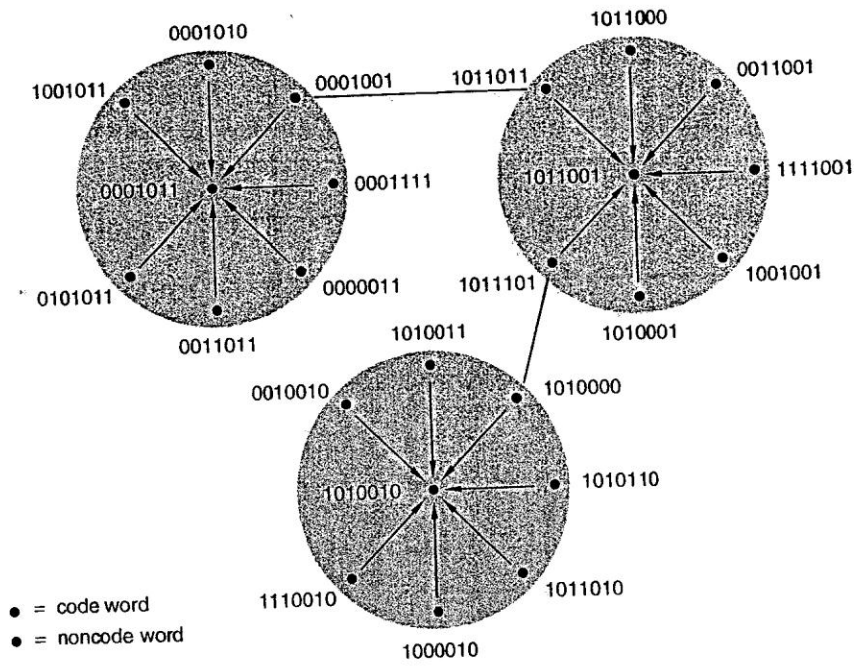

**Figure 2-11** Some code words and noncode words in a 7-bit, distance-3 code.

and assume that failures affect at most one bit of each received code word. Then a received noncode word with a 1-bit error will be closer to the originally transmitted code word than to any other code word. Therefore, when we receive a noncode word, we can *correct* the error by changing the received noncode word to the nearest code word, as indicated by the arrows in the figure. Deciding which code word was originally transmitted to produce a received word is called *decoding*, and the hardware that does this is an error-correcting *decoder*.

*error correction*
*decoding*
*decoder*
*error-correcting code*

A code that is used to correct errors is called an *error-correcting code*. In general, if a code has minimum distance $2c + 1$, it can be used to correct errors that affect up to $c$ bits ($c = 1$ in the preceding example). If a code's minimum distance is $2c + d + 1$, it can be used to correct errors in up to $c$ bits and to detect errors in up to $d$ additional bits.

For example, Figure 2-12(a) on the next page shows a fragment of the $n$-cube for a code with minimum distance 4 ($c = 1$, $d = 1$). Single-bit errors that produce noncode words 00101010 and 11010011 can be corrected. However, an error that produces 10100011 cannot be corrected, because no single-bit error can produce this noncode word, and either of two 2-bit errors could have produced it. So the code can detect a 2-bit error, but it cannot correct it.

When a noncode word is received, we don't know which code word was originally transmitted; we only know which code word is closest to what we've received. Thus, as shown in Figure 2-12(b), a 3-bit error may be "corrected" to the wrong value. The possibility of making this kind of mistake may be acceptable if 3-bit errors are very unlikely to occur. On the other hand, if we are concerned about 3-bit errors, we can change the decoding policy for the code. Instead of trying to correct errors, we just flag all noncode words as uncorrectable errors. Thus, as shown in (c), we can use the same distance-4 code to detect up to 3-bit errors but correct no errors ($c = 0$, $d = 3$).

### 2.15.3 Hamming Codes

*Hamming code*

In 1950, R. W. Hamming described a general method for constructing codes with a minimum distance of 3, now called *Hamming codes*. For any value of $i$, his method yields a $(2^i - 1)$-bit code with $i$ check bits and $2^i - 1 - i$ information bits. Distance-3 codes with a smaller number of information bits are obtained by deleting information bits from a Hamming code with a larger number of bits.

*parity-check matrix*

The bit positions in a Hamming code word can be numbered from 1 through $2^i - 1$. In this case, any position whose number is a power of 2 is a check bit, and the remaining positions are information bits. Each check bit is grouped with a subset of the information bits, as specified by a *parity-check matrix*. As

**DECISIONS, DECISIONS** The names *decoding* and *decoder* make sense, since they are just distance-1 perturbations of *deciding* and *decider*.

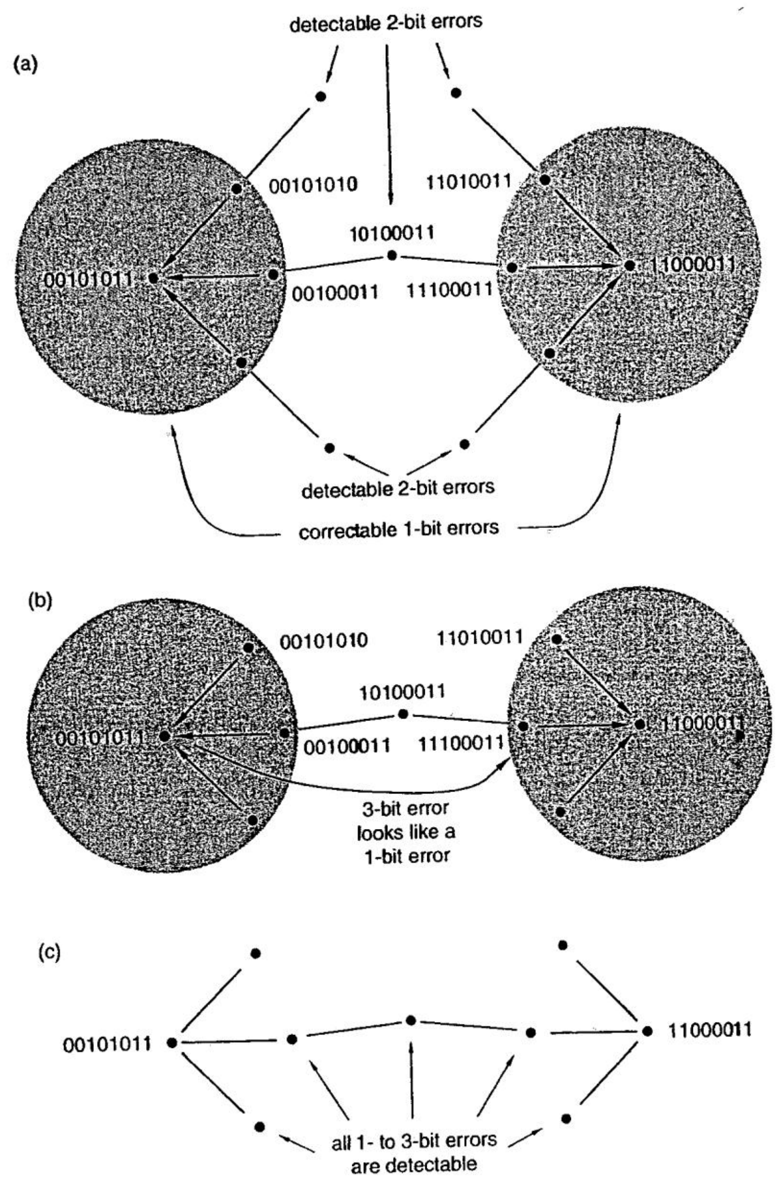

**Figure 2-12**
Some code words and noncode words in an 8-bit, distance-4 code:
(a) correcting 1-bit and detecting 2-bit errors;
(b) incorrectly "correcting" a 3-bit error;
(c) correcting no errors but detecting up to 3-bit errors.

shown in Figure 2-13(a), each check bit is grouped with the information positions whose numbers have a 1 in the same bit when expressed in binary. For example, check bit 2 (010) is grouped with information bits 3 (011), 6 (110), and 7 (111). For a given combination of information-bit values, each check bit is chosen to produce even parity, that is, so the total number of 1s in its group is even.

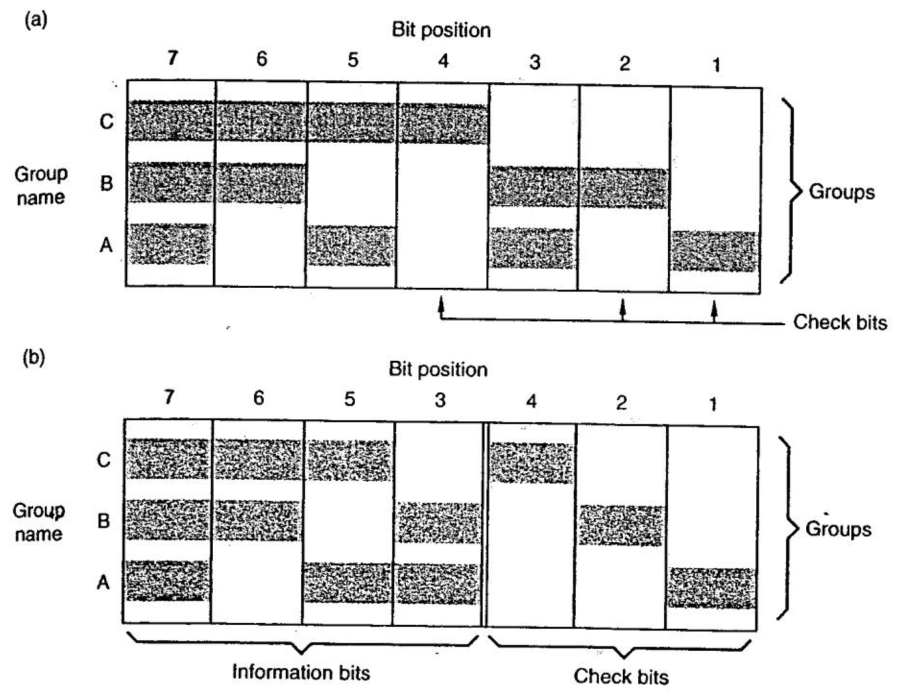

**Figure 2-13**
Parity-check matrices for 7-bit Hamming codes:
(a) with bit positions in numerical order;
(b) with check bits and information bits separated.

Traditionally, the bit positions of a parity-check matrix and the resulting code words are rearranged so that all of the check bits are on the right, as in Figure 2-13(b). The first two columns of Table 2-14 list the resulting code words.

We can prove that the minimum distance of a Hamming code is 3 by proving that at least a 3-bit change must be made to a code word to obtain another code word. That is, we'll prove that a 1-bit or 2-bit change in a code word yields a noncode word.

If we change one bit of a code word, in position *j*, then we change the parity of every group that contains position *j*. Since every information bit is contained in at least one group, at least one group has incorrect parity, and the result is a noncode word.

What happens if we change two bits, in positions *j* and *k*? Parity groups that contain both positions *j* and *k* will still have correct parity, since parity is not affected when an even number of bits are changed. However, since *j* and *k* are different, their binary representations differ in at least one bit, corresponding to one of the parity groups. This group has only one bit changed, resulting in incorrect parity and a noncode word.

If you understand this proof, you should also understand how the position-numbering rules for constructing a Hamming code are a simple consequence of the proof. For the first part of the proof (1-bit errors), we required that the position numbers be nonzero. And for the second part (2-bit errors), we required

**Table 2-14** Code words in distance-3 and distance-4 Hamming codes with four information bits.

| *Minimum-Distance-3 Code* | | *Minimum-Distance-4 Code* | |
|---|---|---|---|
| ***Information Bits*** | ***Parity Bits*** | ***Information Bits*** | ***Parity Bits*** |
| 0000 | 000 | 0000 | 0000 |
| 0001 | 011 | 0001 | 0111 |
| 0010 | 101 | 0010 | 1011 |
| 0011 | 110 | 0011 | 1100 |
| 0100 | 110 | 0100 | 1101 |
| 0101 | 101 | 0101 | 1010 |
| 0110 | 011 | 0110 | 0110 |
| 0111 | 000 | 0111 | 0001 |
| 1000 | 111 | 1000 | 1110 |
| 1001 | 100 | 1001 | 1001 |
| 1010 | 010 | 1010 | 0101 |
| 1011 | 001 | 1011 | 0010 |
| 1100 | 001 | 1100 | 0011 |
| 1101 | 010 | 1101 | 0100 |
| 1110 | 100 | 1110 | 1000 |
| 1111 | 111 | 1111 | 1111 |

that no two positions have the same number. Thus, with an *i*-bit position number, you can construct a Hamming code with up to $2^i - 1$ bit positions.

*error-correcting decoder*

The proof also suggests how we can design an *error-correcting decoder* for a received Hamming code word. First, we check all of the parity groups; if all have even parity, then the received word is assumed to be correct. If one or more groups have odd parity, then a single error is assumed to have occurred. The pattern of groups that have odd parity (called the *syndrome*) must match one of the columns in the parity-check matrix; the corresponding bit position is assumed to contain the wrong value and is complemented. For example, using the code defined by Figure 2-13(b), suppose we receive the word 0101011. Groups B and C have odd parity, corresponding to position 6 of the parity-check matrix (the syndrome is 110, or 6). By complementing the bit in position 6 of the received word, we determine that the correct word is 0001011.

*syndrome*

*extended Hamming code*

A distance-3 Hamming code can easily be extended to increase its minimum distance to 4. We simply add one more check bit, chosen so that the parity of all the bits, including the new one, is even. As in the 1-bit even-parity code, this bit ensures that all errors affecting an odd number of bits are detectable. In particular, any 3-bit error is detectable. We already showed that 1- and 2-bit errors are detected by the other parity bits, so the minimum distance of the extended code must be 4.

Distance-3 and distance-4 extended Hamming codes are commonly used to detect and correct errors in computer memory systems, especially in large servers where memory circuits account for the bulk of the system's electronics and hence failures. These codes are especially attractive for very wide memory words, since the required number of parity bits grows slowly with the width of the memory word, as shown in Table 2-15.

**Table 2-15** Word sizes of distance-3 and distance-4 extended Hamming codes.

| | *Minimum-Distance-3 Codes* | | *Minimum-Distance-4 Codes* | |
|---|---|---|---|---|
| ***Information Bits*** | ***Parity Bits*** | ***Total Bits*** | ***Parity Bits*** | ***Total Bits*** |
| 1 | 2 | 3 | 3 | 4 |
| ≤ 4 | 3 | ≤ 7 | 4 | ≤ 8 |
| ≤ 11 | 4 | ≤ 15 | 5 | ≤ 16 |
| ≤ 26 | 5 | ≤ 31 | 6 | ≤ 32 |
| ≤ 57 | 6 | ≤ 63 | 7 | ≤ 64 |
| ≤ 120 | 7 | ≤ 127 | 8 | ≤ 128 |

### 2.15.4 CRC Codes

Beyond Hamming codes, many other error-detecting and -correcting codes have been developed. The most important codes, which happen to include Hamming codes, are the *cyclic-redundancy-check (CRC) codes*. A rich theory has been developed for these codes, focused both on their error-detecting and correcting properties and on the design of inexpensive encoders and decoders for them (see References).

*cyclic-redundancy-check (CRC) code*

Two important applications of CRC codes are in disk drives and in data networks. In a disk drive, each block of data (typically 512 bytes) is protected by a CRC code, so that errors within a block can be detected and sometimes corrected. In a data network, each packet of data ends with check bits in a CRC code. The CRC codes for both applications were selected because of their burst-error detecting properties. In addition to single-bit errors, they can detect multibit errors that are clustered together within the disk block or packet. Such errors are more likely than errors of randomly distributed bits, because of the likely physical causes of errors in the two applications—surface defects in disk drives and noise bursts in communication links.

## 2.15.5 Two-Dimensional Codes

*two-dimensional code*

Another way to obtain a code with large minimum distance is to construct a *two-dimensional code*, as illustrated in Figure 2-14(a). The information bits are conceptually arranged in a two-dimensional array, and parity bits are provided to check both the rows and the columns. A code $C_{\text{row}}$ with minimum distance $d_{\text{row}}$ is used for the rows, and a possibly different code $C_{\text{col}}$ with minimum distance $d_{\text{col}}$ is used for the columns. That is, the row-parity bits are selected so that each row is a code word in $C_{\text{row}}$ and the column-parity bits are selected so that each column is a code word in $C_{\text{col}}$. (The "corner" parity bits can be chosen according to either code.) The minimum distance of the two-dimensional code is the product of $d_{\text{row}}$ and $d_{\text{col}}$; in fact, two-dimensional codes are sometimes called *product codes*.

*product code*

As shown in Figure 2-14(b), the simplest two-dimensional code uses 1-bit even-parity codes for the rows and columns and has a minimum distance of $2 \cdot 2$, or 4. You can easily prove that the minimum distance is 4 by convincing yourself that any pattern of one, two, or three bits in error causes incorrect parity of a row or a column or both. In order to obtain an undetectable error, at least four bits must be changed in a rectangular pattern as in (c).

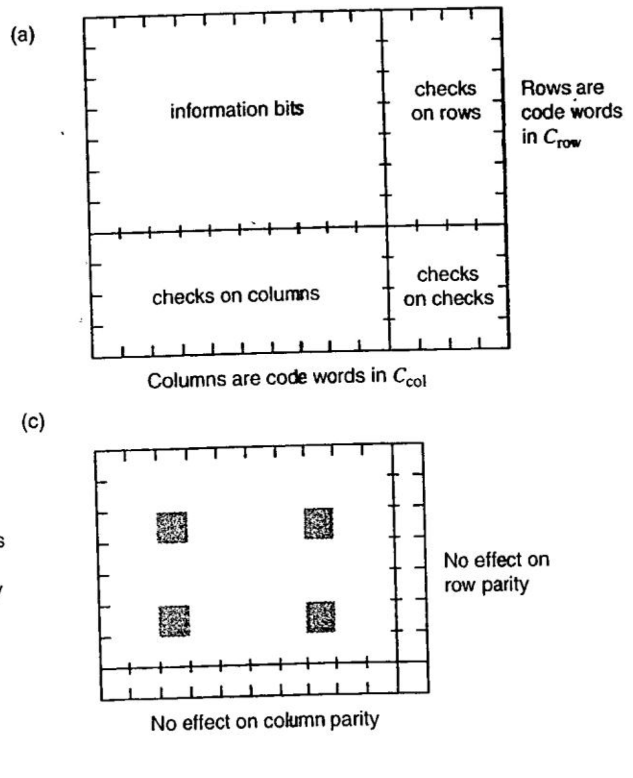
(a)
information bits
checks on rows
Rows are code words in $C_{\text{row}}$
checks on columns
checks on checks
Columns are code words in $C_{\text{col}}$
(b)
information bits
Rows are code words in 1-bit even-parity code
Columns are code words in 1-bit even-parity code
(c)
No effect on row parity
No effect on column parity

**Figure 2-14** Two-dimensional codes: (a) general structure; (b) using even parity for both the row and column codes to obtain minimum distance 4; (c) typical pattern of an undetectable error.

The error-detecting and correcting procedures for this code are straightforward. Assume we are reading information one row at a time. As we read each row, we check its row code. If an error is detected in a row, we can't tell which bit is wrong from the row check alone. However, assuming only one row is bad, we can reconstruct it by forming the bit-by-bit Exclusive OR of the columns, omitting the bad row, but including the column-check row.

To obtain an even larger minimum distance, a distance-3 or -4 Hamming code can be used for the row or column code or both. It is also possible to construct a code in three or more dimensions, with minimum distance equal to the product of the minimum distances in each dimension.

*RAID*

An important application of two-dimensional codes is in RAID storage systems. *RAID* stands for "redundant array of inexpensive disks." In this scheme, $n+1$ identical disk drives are used to store $n$ disks worth of data. For example, eight 200-gigabyte drives could be used to store 1.6 terabytes of nonredundant data, and a ninth 200-gigabyte drive would be used to store checking information. With such a setup, you could store about 400 movies in MPEG-2 format, and never have to worry about losing one to a (single) hard-drive crash!

Figure 2-15 shows the general scheme of a two-dimensional code for a RAID system; each disk drive is considered to be a row in the code. Each drive stores $m$ blocks of data, where a block typically contains 512 bytes. For example, a 200-gigabyte drive would store about 4 billion blocks. As shown in the figure, each block includes its own check bits in a CRC code, to detect and possibly correct errors within that block. The first $n$ drives store the nonredundant data. Each block in drive $n+1$ stores parity bits for the corresponding blocks in the first $n$

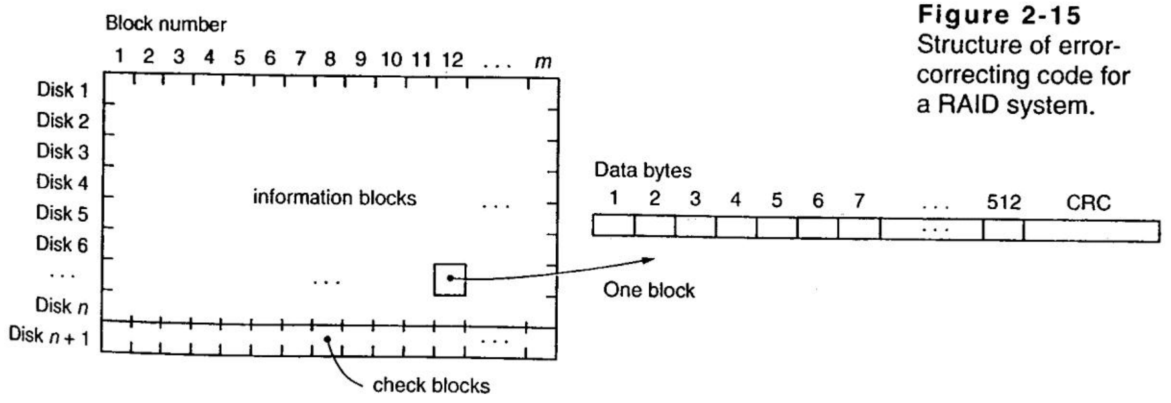

**Figure 2-15** Structure of error-correcting code for a RAID system.

drives. That is, each bit $i$ in drive $n+1$, block $b$, is chosen so that there are an even number of 1s in block $b$, bit position $i$, across all the drives.

In operation, errors in the information blocks are detected by the CRC code. Whenever an error is detected on one of the drives and cannot be corrected using the local CRC code, the block's original contents can still be reconstructed by computing the parity of the corresponding blocks in all the other drives, including drive $n+1$. This method still works even if you lose *all* of the data on a single drive.

Although RAID correction operations require $n$ extra disk reads plus some computation, it's better than losing your data! Write operations also have extra disk accesses, to update the corresponding check block when an information block is written (see Exercise 2.50). Since disk writes are much less frequent than reads in typical applications, this overhead usually is not a problem.

### 2.15.6 Checksum Codes

The parity-checking operation that we've used in the previous subsections is essentially modulo-2 addition of bits—the sum modulo 2 of a group of bits is 0 if the number of 1s in the group is even, and 1 if it is odd. This approach of modular addition can be extended to other bases besides 2 to form check digits.

For example, a computer stores information as a set of 8-bit bytes. Each byte may be considered to have a decimal value from 0 to 255. Therefore, we can use modulo-256 addition to check the bytes. We form a single check byte, called a *checksum*, that is the sum modulo 256 of all the information bytes. The resulting *checksum code* can detect any single *byte* error, since such an error will cause a recomputed sum of bytes to disagree with the checksum.

*checksum*
*checksum code*

Checksum codes can also use a different modulus of addition. In particular, checksum codes using modulo-255, ones'-complement addition are important because of their special computational and error-detecting properties, and because they are used to check packet headers in the ubiquitous Internet Protocol (IP) (see References).

*ones'-complement checksum code*

### 2.15.7 *m*-out-of-*n* Codes

The 1-out-of-$n$ and $m$-out-of-$n$ codes that we introduced in Section 2.13 have a minimum distance of 2, since changing only one bit changes the total number of 1s in a code word and therefore produces a noncode word.

These codes have another useful error-detecting property—they detect unidirectional multiple errors. In a *unidirectional error*, all of the erroneous bits change in the same direction (0s change to 1s, or vice versa). This property is very useful in systems where the predominant error mechanism tends to change all bits in the same direction.

*unidirectional error*