1 Error Correction and Detection

A variety of factors such as faulty components and inadequate design tolerances can result in errors appearing in the information being processed by a computer. Such errors frequently occur in information transmission between two relatively distant points.

1.1 Theory

Two kinds of defenses can be envisioned against such errors. The minimal defense would be Error Detection which only aims at detecting whether an error has occurred or not. Error Correction in contrast aims at correcting the error in case it occurs, thus restoring the representation back to its correct form.

It can be easily seen that an irredundant representation would be unable to deliver either of these attributes as an incorrect word in an irredundant representation would look as if it were a correct one. Redundancy is key in terms of being able to deliver these fault resilience properties. A regular way of delivering such redundancy properties can be conceptualized in terms of Hamming distances.

Hamming distance is a formal denotation of the minimum distance between any two words in the system. It can be seen that an irredundant representation has a Hamming distance of 1 as any two words can differ in as little as 1 bit. A representational scheme that ensures a Hamming distance of 2 on the other hand can be seen to deliver error detection capabilities against a single error, since a flip of a bit would result in a word that is clearly incorrect, as it will take a minimum of 2 flips to reach another correct word.

Analogously, in order to detect up to \( n \) bit errors, a code with Hamming distance of \( n + 1 \) is required. With such a code, no \( n \) bit flip can transform a correct word to another seemingly correct word, as the nearest such would be \( n + 1 \) flips away. On the other hand, correcting \( n \) bit errors would require a Hamming distance of \( 2n + 1 \). An \( n \) bit flip in this case would fail to reach the half way point between two correct words, thus avoiding ambiguity.

Of course, the treatment heretofore only addresses the theoretical fundamentals of error detection and correction and is moot on the issue of how to implement such codes. We turn our attention now to the issues of implementation.
1.2 Implementation

One of the simplest and most widely used techniques for error control is using a single *parity bit* \( C_0 \). This parity bit is appended to an \( n \) bit input word \( X = (X_0X_1...X_{n-1}) \) to form an \( (n+1) \) bit word \( X^* = (X_0X_1...X_{n-1}C_0) \). The value assigned to bit \( C_0 \) makes the total number of bits in \( X^* \) even (if we are using even parity) or odd (in the case of odd parity). Thus, at the receiving end, an error can be detected by taking an *exclusive-or* of all the bits received \( (X^*) \).

Note that this simple technique of error control allows only detection of an error. It is not possible to determine the bit in error (or the position of the error). Hence, it is not possible to correct the error at the receiving end. This leads to retransmission of the information. In addition, this parity only provides single error detection (or in fact, an odd number of errors can be detected). It does not detect multiple errors (even in number).

The parity-checking concept can be easily extended to detection of multiple errors, or to the location of single or multiple errors. These goals can be achieved by providing additional parity bits, each of which check the parity of a subset of the bits in \( X^* \). For example, if we can deduce (from the parity bits) that bit \( X_i \) is in error, then the error can be corrected by simply complementing that bit, thus proving single error correction. Let \( c \) be the number of parity bits required to achieve single error correction with \( n \)-bit data words. Clearly, the check bits must be able to distinguish between \( (n+c) \) possible error location and the single error-free case. Hence,

\[
2^c \geq n + c + 1
\]

It can be proved that this code is *Single Error Correcting Double Error Detecting (SECDED)* code. This is also known as *Hamming Code*.

The key question that remains to be answered is: How to generate the value of \( c \) parity bits that are appended to \( X = (x_0x_1...x_{n-1}) \) to obtain \( X^* \)? Also, how to obtain the location of error from \( X^* \)? The process is illustrated in the following example:

Let \( n = 4 \). Then, \( c = 3 \) (since, \( 2^3 \geq 4 + 3 + 1 \)). Let \( M = (m_0m_1m_2m_3) \) be the message bits and let the three parity bits be \( (p_0p_1p_2) \). Organize the bits as follows:

<table>
<thead>
<tr>
<th>POSITION</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>( m_0 )</td>
<td>( m_1 )</td>
<td>( m_2 )</td>
<td>( p_0 )</td>
<td>( m_3 )</td>
<td>( p_1 )</td>
<td>( p_2 )</td>
</tr>
</tbody>
</table>

Then, the check bits \( (p_0p_1p_2) \) can be computed as follows:

\[
p_2 = m_3 \oplus m_2 \oplus m_0 \quad (1)
\]

\[
p_1 = m_3 \oplus m_1 \oplus m_0 \quad (2)
\]

\[
p_0 = m_2 \oplus m_1 \oplus m_0 \quad (3)
\]
Note that one change in data bits produces at least two changes in check bits. Similarly, two changes in data bits produce at least one change in check bits. Hence, the minimum distance between two valid code words is three. (Recall that the distance between two words is equal to the number of bits that need to be flipped to obtain one number from other. For example, the distance between 0110 and 0101 is 2.) As a result, it is a single-error-correcting code.

The position of the single error can be detected using vector \((c_0c_1c_2)\) derived as follows:

\[
c_2 = p_2 \oplus m_3 \oplus m_2 \oplus m_0 \tag{4}
\]

\[
c_1 = p_1 \oplus m_3 \oplus m_1 \oplus m_0 \tag{5}
\]

\[
c_0 = p_0 \oplus m_2 \oplus m_1 \oplus m_0 \tag{6}
\]

Thus, if \(c_0 = c_1 = c_2 = 0\), then there is no single error.

The general technique to obtain these equation is described as follows:

1. Given \(n\) (the number of bits in the message), compute \(c\) using \(2^c \geq n + c + 1\). Let \((n + c)\) be the total number of bits. Label the position of bits from 1 through \((n + c)\) from right to left. For example, for \(n = 4\), we get \(c = 3\). Hence, we label these 7 bit positions from 1 through 7 (from right to left) as follows:

   \[
   \begin{array}{cccccccc}
   \text{POSITION} & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\
   \text{BIT} & m_0 & m_1 & m_2 & p_0 & m_3 & p_1 & p_2 
   \end{array}
   \]

2. Position the \(c\) parity bits in the locations marked with a number that is a power of 2. Place the message bits in other positions.

   Hence, our example becomes as follows:

   \[
   \begin{array}{cccccccc}
   \text{POSITION} & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\
   \text{BIT} & m_0 & m_1 & m_2 & p_0 & m_3 & p_1 & p_2 
   \end{array}
   \]

3. To find the equation for the parity bit in position \(2^i\), do the following:

   - Construct the sequence of numbers 0 through \((n + c)\) in binary.

   Hence, for our example, it is as follows:

   \[
   \begin{array}{c|c}
   0 & 000 \\
   1 & 001 \\
   2 & 010 \\
   3 & 011 \\
   4 & 100 \\
   5 & 101 \\
   6 & 110 \\
   7 & 111 
   \end{array}
   \]
• The equation for parity bit in position $2^i$ is the *exclusive-or* ($\oplus$) of the bits in positions $j$ (except for $j = 2^i$) such that the $i$th bit in the binary code for $j$ (in the table created in the previous step) is equal to 1.

For example, for the parity bit $p_2$, the position = 1. Hence, $2^i = 1 \Rightarrow i = 0$. From the table created in the previous step, the 0th bit in the binary code is 1 for numbers (1, 3, 5, 7). Hence, the equation for $p_2$ is

$$p_2 = m_3 \oplus m_2 \oplus m_0$$

Note that, we do not use the bit in position 1 since $2^i = 1$. 