TEMA I: INTRODUCCIÓN A LA ELECTRÓNICA DIGITAL

1. Electrónica Digital

Antes de empezar en el tema en cuestión, vamos a dar una posible definición de la disciplina que vamos a tratar, así como su ámbito de acción. Una posible definición de Electrónica puede ser *la ciencia que estudia los fenómenos relacionados con el transporte de carga eléctrica en medios materiales junto con la construcción de dispositivos, circuitos y sistemas basados en ellos*. Esta ciencia se divide en dos grandes ramas: Analógica y Digital.

La Electrónica Analógica es la parte de la Electrónica que trabaja con variables continuas de tal forma que un pequeño cambio en alguna variable puede producir un gran cambio en el comportamiento del circuito. Por lo tanto, las variables serán números reales. Un ejemplo de estos circuitos puede ser un amplificador de señal.

La Electrónica Digital es la parte de la Electrónica que trabaja con variables discretas. Este hecho implica que un pequeño cambio en alguna de las variables del circuito (siempre que no cambie su valor discreto) no producirá un cambio apreciable en el comportamiento del circuito. Es decir, el comportamiento del circuito no depende del valor exacto de la señal.

Un mismo circuito electrónico, dependiendo de la zona de operación que interese, tiene su operación analógica y digital. Por ejemplo, en la figura 1.1 mostramos la forma de operación de un sistema electrónico típico. En esta forma de operación, podemos observar que en los extremos apenas existe un cambio en la señal de salida independientemente de la magnitud del cambio en la señal de entrada; estas son las zonas que le interesan a la rama digital de la Electrónica, ya que sin conocer el valor exacto de la señal de entrada podemos dar un valor de salida. En cambio, la parte central se produce el caso contrario, es decir, para un pequeño cambio de la señal de entrada existe un gran cambio en la señal de salida; esta es la zona donde se interesa la rama analógica de la Electrónica, ya que debemos conocer exactamente el valor de la señal de entrada (su valor continuo) para poder dar un valor a la señal de salida.

Una vez llegados a este punto, nos preguntamos por qué estudiar los circuitos digitales. La razón básica radica en el aumento de fiabilidad en el procesamiento y transmisión de la información ya que una pequeña degradación de la señal no influirá en su valor (o en su influencia como entrada en un circuito digital). A este motivo se le pueden añadir otros más, como pueden ser: se dispone de un soporte matemático adecuado, como son las álgebras discretas; existen tecnologías de fabricación adecuadas; contamos con una amplia distribución comercial debido a su amplias aplicaciones en múltiples campos, etc.

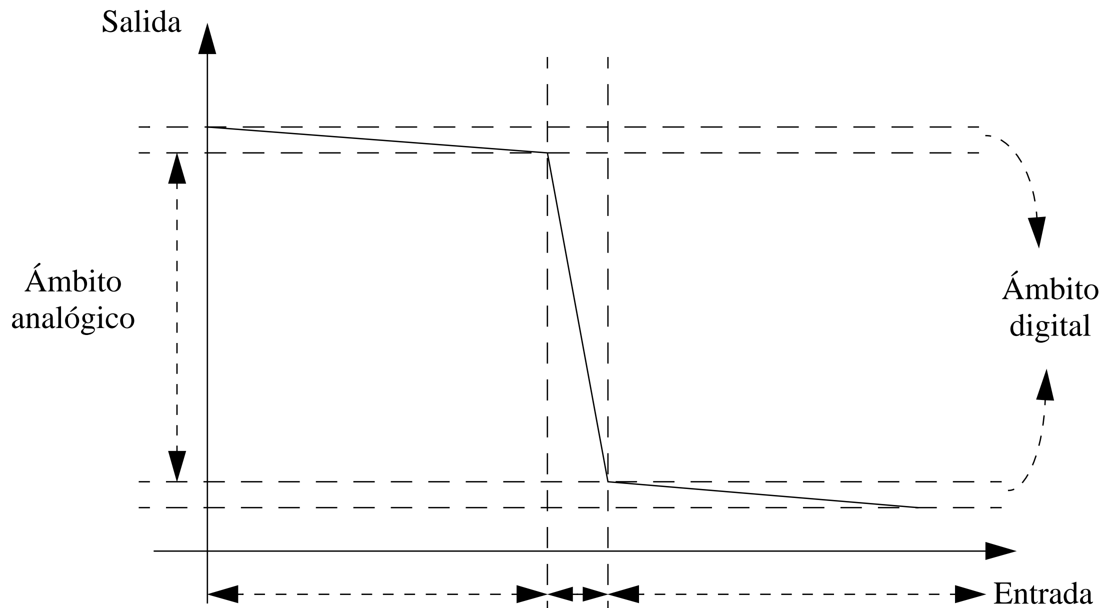


Figura 1.1.- Diferencia existente entre Electrónica Analógica y Electrónica Digital.

Dentro de los circuitos digitales, existe una división en dos grandes grupos: circuitos combinacionales y circuitos secuenciales. Los circuitos combinacionales se caracterizan por el hecho de que las salidas únicamente dependen de la combinación de entradas y no de la historia anterior del circuito; por lo tanto, no tienen memoria y el orden de la secuencia de entradas no es significativo. Los circuitos secuenciales se caracterizan por el hecho de que las salidas dependen de la historia anterior del circuito además de la combinación de entradas; por lo que estos circuitos sí disponen de memoria y el orden de la secuencia de entradas sí es significativo.

2. Representación Numérica

Debido al carácter discreta y a los componentes utilizados en Electrónica Digital (en la actualidad, dispositivos como transistores y diodos), no se va a emplear el sistema decimal sino el sistema binario. Por lo tanto, es necesario introducirnos en la teoría de los sistemas numéricos, con vistas a su aplicación a la conversión entre los principales sistemas.

En cualquier sistema numérico, se define la base o raíz como *el número máximo de dígitos disponibles en dicho sistema*. Así, los sistemas numéricos más importantes son los siguientes:

- Sistema decimal o de base 10, que consta de diez dígitos: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.
- Sistema binario o de base 2, que consta de dos dígitos: {0, 1}.
- Sistema octal o de base 8, que consta de ocho dígitos: {0, 1, 2, 3, 4, 5, 6, 7}.
- Sistema hexadecimal o de base 16, que consta de dieciséis dígitos: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}.

La notación posicional de un número es la colocación de los dígitos del sistema numérico, al que pertenece, uno detrás de otro. El valor numérico correspondiente a un numero N en un sistema de base b es:

$$(N)_b = p_n b^n + p_{n-1} b^{n-1} + \ldots + p_0 b^0 + p_{-1} b^{-1} + \quad + p_{-m} b^{-m} = \sum_{i=-m}^{n} p_i b^i$$

Donde p_i es el dígito correspondiente en dicho sistema, verificando que $0 <= p_i <= b-1$.Esto es debido a que p0 es el resto de la división de N entre b. Por ejemplo:

$$(123.2)_{10} = 1*10^2 + 2*10^1 + 3*10^0 + 2*10^{-1}$$

$$(1211.1)_3 = 1*3^3 + 2*3^2 + 1*3^1 + 1*3^0 + 1*3^{-1}$$

$$(F02.1)_{16} = 15*16^2 + 0*16^1 + 2*16^0 + 1*16^{-1}$$

Como ya hemos mencionado, los sistemas numéricos más importantes son los sistemas decimal, binario, octal y hexadecimal. El sistema decimal es el empleado en la vida cotidiana. El sistema binario es el sistema que emplearemos en Electrónica Digital, ya que trataremos la mayoría de los elementos como interruptores, es decir, con dos estados: encendido (permite el paso de la información, estado ‘1’) o apagado (no permite dicho paso, estado ‘0’). No obstante como $(100)_{10} = (1100100)_2$, observamos que se necesita un número relativamente grande de dígitos binarios para expresar un número decimal relativamente pequeño. Por lo tanto, aparecieron los sistemas octal y hexadecimal para disminuir este número de dígitos sin demasiada complicación puesto que su conversión a binario y viceversa es muy directa como ya veremos.

2.1. Conversiones entre sistemas numéricos.

Para trabajar con varios sistemas numéricos, como por ejemplo el sistema decimal y el binario debemos ser capaces de convertir números de una base a otra. Existen dos métodos de conversión de base: polinómico e iterativo.

El método polinómico consiste en expresar el número de la base fuente como un polinomio y evaluarlo según la aritmética de la base destino. Por ejemplo:

$$(1011)_2 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = (11)_{10}$$

Este método tiene el problema de que si la base destino no es la decimal, tendremos que usar una aritmética diferente a la que normalmente utilizamos. Por lo tanto, este método suele ser usado para convertir números a un sistema decimal.

El método iterativo consiste en ir dividiendo el número (usando la aritmética de la base fuente) por la base destino de tal forma que los restos nos irán dando los dígitos en la nueva base, siendo el más significativo el último dígito obtenido.

$$(p_3*b^3 + p_2*b^2 + p_1*b^1 + p_0*b^0)/b = p_3*b^2 + p_2*b^1 + p_1*b^0 + p_0/b$$

Resto: p_0

$$\text{Cociente: } p_3*b^2 + p_2*b^1 + p_1*b^0$$

Es decir, cuando se realiza la primera división, el resto que nos queda es el dígito menos significativo en la nueva base y el cociente es el número restante. Este proceso se repite hasta que el cociente sea 0. Por ejemplo:

```
          11 |_2
p0 ------> 1   5 |_2
p1 ----------> 1   2 |_2
p2 --------------> 0   1 |_2
p3 ------------------> 1   0
```

Hasta ahora, sólo hemos tenido en cuenta la parte entera de los números. Con el método polinómico, la parte fraccional se trata igual que la parte entera permitiendo que los exponentes de la base sean negativos. Esto nunca nos dará un número mayor que uno ya que los dígitos son menores que la base. Por ejemplo:

$$(0.101)_2 = 1*2^{-1} + 0*2^{-2} + 1*2^{-3} = (0.625)_{10}$$

Con el método iterativo, la parte fraccional debe evaluarse por separado de la parte entera. La parte fraccional se multiplica por la base, la parte entera corresponde al nuevo dígito, mientras que se vuelve a iterar con la parte fraccional restante.

Del mismo modo que antes, vemos que si realizamos la multiplicación en un polinomio genérico obtenemos dicho resultado.

$$(p_{-1}*b^{-1} + p_{-2}*b^{-2} + p_{-3}*b^{-3}) * b = p_{-1} + p_{-2}*b^{-1} + p_{-3}*b^{-2}$$

$$\text{Parte entera: } p_{-1}$$

$$\text{Parte fraccional: } p_{-2}*b^{-1} + p_{-3}*b^{-2}$$

En este último caso, como ya hemos visto, la aritmética empleada es la de la base fuente, por lo que se suele utilizar para la conversión de números decimales a cualquier otra base. Además, como sucede en cualquier sistema numérico, podemos encontrar números con infinitos dígitos “decimales”. A continuación, ilustramos un ejemplo de conversión de un número fraccionario decimal a binario:

$$0.625 * 2 = 1.250$$
$$0.250 * 2 = 0.500$$
$$0.500 * 2 = 1.000$$
$$0.000 * 2 = 0.000$$

Debido a que las bases octales y hexadecimales son potencia de dos ($16 = 2^4$ y $8 = 2^3$), la conversión entre estos sistemas y el binario es muy sencilla. Fijémonos en el valor numérico de números binarios, octales y hexadecimales genéricos:

$$b_6*2^6 + b_5*2^5 + b_4*2^4 + b_3*2^3 + b_2*2^2 + b_1*2^1 + b_0*2^0$$

$$O_2*8^2 + O_1*8^1 + O_0*8^0$$

$$H_1*16^1 + H_0*16^0$$

Por ser las bases octales y hexadecimales potencias de dos, en el número binario podemos agrupar los dígitos en función de estas potencias:

$$(b_6*2^0)*8^2 + (b_5*2^2 + b_4*2^1 + b_3*2^0)*8^1 + (b_2*2^2 + b_1*2^1 + b_0*2^0)*8^0$$

$$(b_6*2^2 + b_5*2^1 + b_4*2^0)*16^1 + (b_3*2^3 + b_2*2^2 + b_1*2^1 + b_0*2^0)*16^0$$

Al comparar las dos últimas expresiones con las expresiones genéricas de los números octales y hexadecimales, podemos obtener las siguientes igualdades:

$$O_2 = b_6*2^0$$

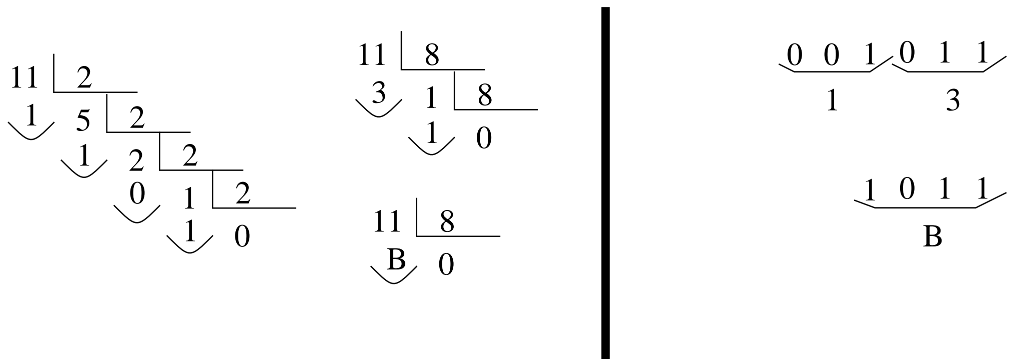
$$O_1 = b_5*2^2 + b_4*2^1 + b_3*2^0$$

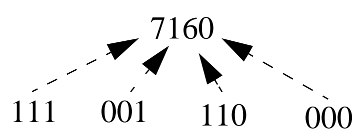
$$O_0 = b_2*2^2 + b_1*2^1 + b_0*2^0$$

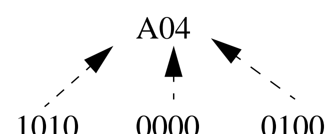
$$H_1 = b_6*2^2 + b_5*2^1 + b_4*2^0$$

$$H_0 = b_3*2^4 + b_2*2^2 + b_1*2^1 + b_0*2^0$$

Por lo tanto, podemos realizar la conversión dígito a dígito, y no tener que tratar con el número completo. Luego cada dígito octal (hexadecimal) será sustituido por grupos de tres (cuatro) dígitos binarios con el mismo valor numérico. En el caso de la conversión desde binario a octal (hexadecimal) consiste en la conversión en grupos de tres (cuatro) dígitos (si faltan a la izquierda dígitos para completar algún grupo, se le añaden 0's). Por ejemplo:







3. Códigos Binarios

A partir de ahora (salvo que se indique lo contrario) vamos a trabajar con el sistema binario, cuyos dígitos se denominan bits. Una vez visto como podemos pasar de una base a otra, nos falta determinar la relación biunívoca existente entre la información y la palabra de código por la que será representada. Esta relación viene dada por el código. Como el sistema numérico con el que trabajamos habitualmente es el sistema decimal, la mayoría de los códigos que veremos serán códigos BCD (decimal codificado en binario). Estos códigos necesitan cuatro bits por cada dígito decimal ya que $2^3 < 10 < 2^4$. Así, el código BCD natural es aquel que representa cada dígito decimal por su expresión polinómica.

Dentro de los códigos, podemos encontrar a un grupo cuyo valor se puede obtener a través de un polinomio con los dígitos de dicho código a través de un producto escalar.

$$\text{Valor} = w_3c_3 + w_2c_2 + w_1c_1 + w_0c_0$$

donde el vector ($w_3w_2w_1w_0$) es conocido como vector peso y el ($c_3c_2c_1c_0$). Los códigos que muestran esta característica se denominan **códigos pesados o con peso**. El caso del código BCD natural se encuentra dentro del grupo de los códigos con peso ya que cada bit se encuentra multiplicado por un peso (8,4,2,1). Otros ejemplos de códigos con peso son los que tienen como peso (8,4, -2, -1) y (2,4,2,1). Un ejemplo de código no pesado es el código de exceso a 3. Este código se obtiene como el código binario del resultado de la suma del dígito en cuestión con el número tres. Estos códigos los podemos ver en la siguiente tabla.

Dígito Decimal	BCD natural												Exceso a 3			
	8	4	2	1	8	4	-2	-1	2	4	2	1				
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	1	1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	0	1	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	1	1	1	0	1	1	1	0	0	0
6	0	1	1	0	1	0	1	0	1	1	0	0	1	0	0	1
7	0	1	1	1	1	0	0	1	1	1	0	1	1	0	1	0
8	1	0	0	0	1	0	0	0	1	1	1	0	1	0	1	1
9	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0

Tabla 1.1. Ejemplos de códigos binarios.

Excepto el código BCD natural, los otros tres tienen la propiedad de ser **autocomplementarios**, es decir, si complementamos (cambiar 1's por 0's y viceversa) la palabra bit a bit, obtendremos la palabra correspondiente al dígito 9-d.

En algunos dispositivos es útil que las palabras correspondientes a valores próximos se diferencien poco para que el posible error en la transmisión no engañe demasiado el resultado final. Por ejemplo, si pasamos de 7 (0111) a 8 (1000) y se produce un fallo en el último bit, de tal forma que no se produce su transición, el resultado obtenido sería de 0000 (0). Así, se define la **distancia de Hamming** como *el número de bits que difieren entre dos palabras de código*; y se define un código de distancia n como aquel cuya distancia de palabras consecuti-

vas es n. Estos códigos se pueden clasificar en: códigos cerrados, cuando la última palabra tiene una distancia n con respecto a la primera; y códigos abiertos, cuando la última palabra tiene una distancia diferente a n con respecto a la primera palabra. Un ejemplo de código de distancia unidad cerrado es el código de Gray, que se caracteriza con una fácil implementación a partir del código binario.

Dígito Decimal	Binario natural				Código Gray				
	b3	b2	b1	b0	g3	g2	g1	g0	
0	0	0	0	0	0	0	0	0	
1	0	0	0	1	0	0	0	1	
2	0	0	1	0	0	0	1	1	
3	0	0	1	1	0	0	1	0	
4	0	1	0	0	0	1	1	0	$g_i = 0 \Leftrightarrow b_i = b_{i+1}$
5	0	1	0	1	0	1	1	1	
6	0	1	1	0	0	1	0	1	
7	0	1	1	1	0	1	0	0	
8	1	0	0	0	1	1	0	0	
9	1	0	0	1	1	1	0	1	

Tabla 1.2. Generación del código Gray a partir del código binario natural.

3.1. Códigos detectores de error.

A la hora de utilizar diferentes códigos, podemos encontrar diversos motivos, como puede ser evitar que la información caiga en manos no deseadas. No obstante, otro motivo más práctico consiste en enviar información sobre la existencia de posibles fallos en la transmisión. Existen códigos que incluyen información para poder detectar y/o corregir estos posibles fallos.

Una vez llegados a este punto, nos deberemos preguntar sobre cuántos bits consideraremos los fallos. En principio, sería lógico pensar que consideraríamos los fallos de transmisión en todos los bits; pero esta situación sería excesivamente costosa. Luego nos fijaremos en las probabilidades que existen de que ocurran errores según el número de bits erróneos. Según estos estudios, la probabilidad cae excesivamente cuando los fallos ocurren en dos o más bits; por lo tanto, sólo tendremos en cuenta fallos en un solo bit.

Para que sea factible la detección de fallos, se debe garantizar que la palabra obtenida tras el fallo no pertenezca al código en cuestión. Si hablamos en términos de la distancia de Hamming, la palabra correcta y la errónea tienen una distancia igual a “1”. Luego una condición necesaria y suficiente será que todas las palabras del código tengan una distancia mínima de dos entre todas ellas. Así, podemos comprobar la siguiente afirmación:

Cualquier código de detección de fallo en un solo bit debe tener al menos una distancia de dos entre cualesquiera dos palabra de código.

Las palabras con una distancia mínima de dos se pueden usar como códigos de detección de error en un bit, y así poder utilizarlas como chequeadoras del canal de transmisión.

Al tener una distancia mínima de dos, entre dos palabras correctas siempre habrá una combinación de bits (que corresponderá a las posibles palabras erróneas) las cuales no serán ninguna de

nuestras palabras de código. Por lo tanto, cuando se produce un fallo en un solo bit, la combinación obtenida será alguna de las que no pertenecen al código pudiendo detectar el fallo.

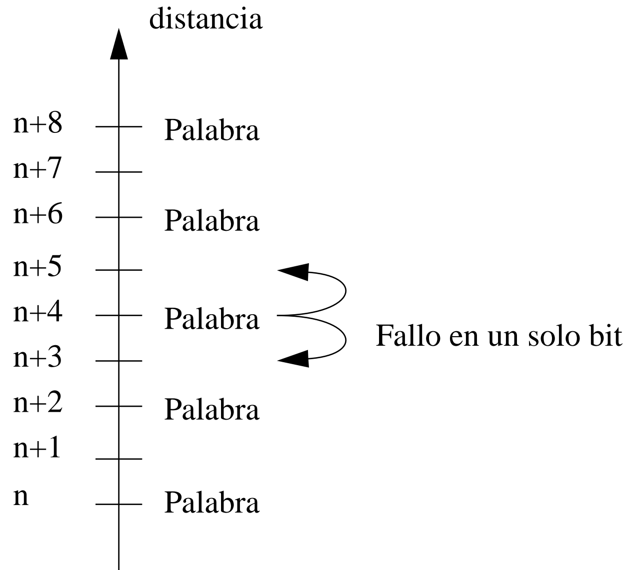


Figura 1.2.- Comportamiento de un código detector de fallos en un solo bit ante la existencia de un fallo en un bit.

Los códigos que solamente detectan la existencia de un fallo se denominan códigos detectores de errores. Dos de los códigos que puede detectar errores son los códigos “2-out-of-5” y biquinario.

El código 2-out-of-5 tiene la peculiaridad de que tiene todas las combinaciones de cinco bits que tienen dos 1’s entre todos sus bits. Por lo tanto, una forma de detectar errores será la comprobación de que únicamente existan dos 1’s. De esta forma, un error en un bit nos aportará una palabra con tres 1’s o ningún 1. La detección no se da para fallos en más bits por dos razones:

- Fallos en un número par de bits puede producir una palabra con dos 1’s, considerada correcta.
- Fallos en un número impar de bits puede producir el mismo comportamiento que el fallo en un solo bit, no pudiendo saber en cuántos bits se ha producido el fallo.

El código biquinario muestra un aspecto similar al anterior, es decir, todas sus palabras contienen dos 1’s en todos sus bits, pero a diferencia del primero no contiene todas las combinaciones de dos 1’s en todos sus bits, no obstante existe una relación que sólo es cumplida por las palabras correctas del código. Así la detección de un fallo se realiza comprobando que exista un 1 en los dos primeros bits, y otro 1 en los cinco restantes. Al igual que el anterior sólo se considera fallos en un solo bit por razones similares. Podemos comprobar, que este código es pesado, propiedad que no comparte con el anterior.

No obstante, la técnica más empleada es la adición de un nuevo bit, denominado bit de paridad, a cualquier código. Este bit informará si el número de 1’s de la palabra es par (bit de paridad a 0) o impar (bit de paridad a 1). Por lo tanto, si se produce un fallo en un bit, la información de la paridad será errónea.

Dígito decimal	2-out-of-5	Biquinario	Binario con paridad
		5 0 4 3 2 1 0	b3 b2 b1 b0 p
0	0 0 0 1 1	0 1 0 0 0 0 1	0 0 0 0 0
1	0 0 1 0 1	0 1 0 0 0 1 0	0 0 0 1 1
2	0 0 1 1 0	0 1 0 0 1 0 0	0 0 1 0 1
3	0 1 0 0 1	0 1 0 1 0 0 0	0 0 1 1 0
4	0 1 0 1 0	0 1 1 0 0 0 0	0 1 0 0 1
5	0 1 1 0 0	1 0 0 0 0 0 1	0 1 0 1 0
6	1 0 0 0 1	1 0 0 0 0 1 0	0 1 1 0 0
7	1 0 0 1 0	1 0 0 0 1 0 0	0 1 1 1 1
8	1 0 1 0 0	1 0 0 1 0 0 0	1 0 0 0 1
9	1 1 0 0 0	1 0 1 0 0 0 0	1 0 0 1 0

Tabla 1.3. Ejemplos de códigos detectores de errores en un solo bit.

3.2. Códigos correctores de errores.

Una vez que hemos visto como se pueden detectar errores en la transmisión de la información, debemos pasar a ver como podemos determinar cuál de los bits es el que ha producido el error. Estos códigos se denominan códigos correctores de error.

En el caso de corrección, tenemos que conseguir que la palabra errónea se encuentre más cerca (en el sentido de la distancia de Hamming), o lo que es lo mismo, tenga menos bits diferentes con respecto a una sola palabra del código. Con este criterio de cercanía, podemos asegurar cuál es la palabra que se ha querido transmitir (con mayor probabilidad), y por lo tanto, saber cuál ha sido el bit que ha producido el error.

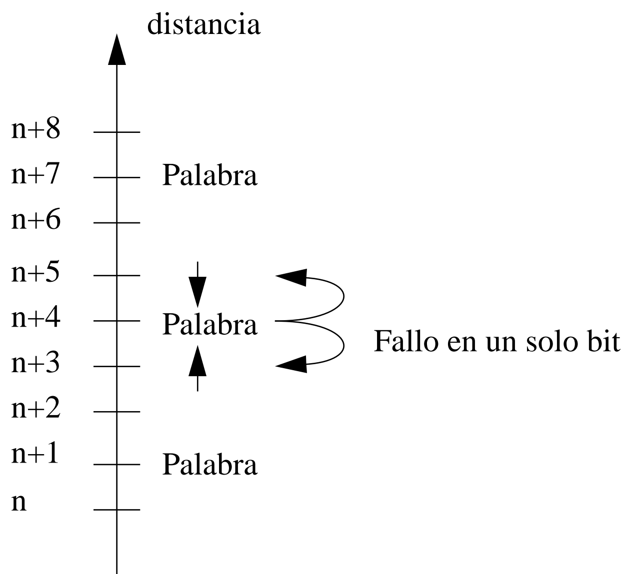


Figura 1.3.- Comportamiento de un código corrector de fallos en un solo bit ante la existencia de un fallo en un bit.

Por lo tanto, la distancia mínima que tiene que tener todas las palabras de código entre sí es igual a tres.

Así, podemos dar una condición necesaria y suficiente para que un código sea corrector en un solo bit:

> *La condición necesaria y suficiente para que cualquier conjunto de palabras binarias sea un código corrector de un error en un solo bit, es que la distancia mínima entre ellas sea de tres.*

Para lograr esta meta se hace necesario añadir más bits (no es suficiente con un solo bit de paridad), por lo que en la palabra podemos diferenciar bits de mensaje (son los que realmente llevan la información útil) y bits de chequeo (son los que llevan la información para corregir los posibles fallos). Estos bits de chequeo suelen ser, por lo general, bits de paridad de un determinado subconjunto de bits de mensaje. Para que se logre este cometido, un fallo en un bit de mensaje debe producir una contradicción en más de un bit de chequeo, mientras que un fallo en un bit de chequeo sólo debe producir una contradicción en dicho bit (bits de chequeo independientes explícitamente). Un ejemplo de estos códigos consta de tres bits de chequeo (C1, C2 y C4) y cuatro bits de mensaje (M3, M5, M6 y M7), mostrado en la tabla 1.4.

Dígito decimal	C1	C2	M3	C4	M5	M6	M7
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

Tabla 1.4. Ejemplo de código corrector de error en un solo bit.

Una vez que disponemos de un código corrector, tenemos que obtener cuáles son los bits de mensaje cuya paridad es almacenada en cada bit de chequeo. Una forma sencilla de obtener esta información consiste en considerar todas las palabras que tengan un solo 1 en todos los bits de mensaje; por lo que los bits de chequeo que tomen el valor 1, "verán" a este bit de mensaje. Por ejemplo, consideremos la palabra "1" que sólo tiene un 1 en el bit M7. En este caso, todos los bits de chequeo tienen el valor 1, por lo que todos ellos chequearán a dicho bit de mensaje. Si repetimos este proceso tantas veces como sea necesaria, obtenemos que:

- C1 es la paridad de M3, M5 y M7
- C2 es la paridad de M3, M6 y M7
- C4 es la paridad de M5, M6 y M7

Veamos ahora como se detecta el bit donde se ha producido un fallo. No podemos olvidar que estamos ante situaciones de errores en un solo bit; así que si detectamos un fallo en dos bits de chequeo, el fallo deberá estar en el único bit de mensaje común a dichos bits de chequeo. Por lo tanto:

- Si existe un fallo en M3, se detecta en C1 y C2
- Si existe un fallo en M5, se detecta en C1 y C4

- Si existe un fallo en M6, se detecta en C2 y C4
- Si existe un fallo en M7, se detecta en C1, C2 y C4
- Si existe un fallo en C1, se detecta en C1
- Si existe un fallo en C2, se detecta en C2
- Si existe un fallo en C4, se detecta en C4

Para poder obtener el número de bits de chequeo para crear un código corrector de errores en un solo bit, se verifica que $k+m<=2^k-1$; donde k es el número de bits de chequeo y m es el número de bits de mensaje. El hecho de 2^k-1 radica en que el número de bits de chequeo debe ser tal que tenga tantas combinaciones como posibilidades de fallos, eliminando la posibilidad de la no existencia de fallo.