Universidad de Granada

Ingeniería Informática

Computación y Sistemas Inteligentes

Practica 3

Autor: José Antonio Ruiz Millán
Juan Carlos Ruiz García
Email: jantonioruiz@correo.ugr.es
jcarlosruiz95@correo.ugr.es
Asignatura: Teoría de la Información y la Codificación
27 de agosto de 2019

1. **(0,5 puntos) Explique cómo diseñar un código de Hamming (7, 4). Dónde se encuentran los bits de paridad y cómo se calcula su valor**

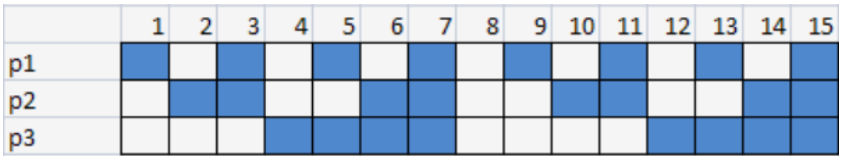
   Para **diseñar un código Hamming (7,4)** lo primero que tenemos que saber es que ésto nos dice que tenemos palabras de $n = 7$ bits, de los cuales 4 bits son de datos, por lo que 3 son de paridad par.

   Para explicar los siguientes pasos, primero voy a comentar dónde podemos encontrar los bits de paridad y cómo calcular su valor.

   Los **bits de paridad se encuentran** en las posiciones que son potencia de 2 dentro de la cadena de bits, por lo que en este caso como tenemos una cadena con 7 bits, los bits de paridad serán los bits de las posiciones 1,2 y 4, teniendo entonces:

   $$(x_4x_3x_2p_3x_1p_2p_1)$$

   Para el **cálculo de los bits de paridad** tenemos que obtener las ecuaciones que dependan únicamente de los valores de datos que son los que conocemos. Por lo que sabiendo que los bits de paridad afectan de la siguiente forma:

   Teniendo esto podemos obtener las distintas ecuaciones y calcular los distintos valores de los bits de paridad. Por lo que entonces tenemos que:

   - $p_3 = x_4 + x_3 + x_2$
   - $p_2 = x_4 + x_3 + x_1$
   - $p_1 = x_4 + x_2 + x_1$

   Una vez tenemos esto calculado, podemos proseguir con el proceso de **creación del código**. Para ello, el siguiente paso es la creación de la matriz del código que nos permite dado un código de datos (4 bits) crear el correspondiente código de Hamming. Por lo que partimos de lo siguiente:

   $$(x_4x_3x_2x_1) \cdot G = (x_4x_3x_2p_3x_1p_2p_1)$$

   Por lo que lo que tenemos que hacer es hallar la matriz G que nos transforma este código. Como sabemos que las posiciones de los bits 3º,5º,6º y 7º son los bits del código, podemos partir de la siguiente matriz:

   $$G = \begin{pmatrix} 1 & 0 & 0 & ? & 0 & ? & ? \\ 0 & 1 & 0 & ? & 0 & ? & ? \\ 0 & 0 & 1 & ? & 0 & ? & ? \\ 0 & 0 & 0 & ? & 1 & ? & ? \end{pmatrix}$$

   Ahora solo nos quedaría rellenar con los valores que pertenecen a los bits de paridad. Estos valores los hemos calculado anteriormente con las ecuaciones, por lo que lo único que tenemos que hacer es poner cada uno de ellos en la columna a la que pertenece, obteniendo finalmente:

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Por lo que con esto ya tenemos la matriz generadora del código de Hamming (7,4) y podemos transformar un código del tipo $(x_4x_3x_2x_1)$ en $(x_4x_3x_2p_3x_1p_2p_1)$.

2. **(0.5 puntos) Explique, dado un código de bloque de Hamming (7,4), cómo sabemos cuántos errores podemos detectar, y qué procedimiento hay que seguir para detectarlos.**

   Por definición, un código de Hamming sólo pueden utilizarse para detectar errores de 2 bits y para corregir errores de 1 único bits.

   No obstante, la forma que podemos utilizar para saber el número de errores que podemos detectar es usando **la distancia de Hamming**. Con la matriz generadora del código de Hamming, calculamos la distancia de Hamming entre todos los elementos y podremos comprobar que siempre obtenemos una distancia mínima de Hamming de 3. Sabiendo que:

   - Para detectar errores de **d** bits, necesitamos al menos una distancia de Hamming de **d+1**.
   - Para corregir errores de **d** bits, necesitamos al menos una distancia de Hamming de **2*d+1**

   Como tenemos una distancia de Hamming de $d + 1 = 3$ y $2 \cdot 1 + 1 = 3$, podemos decir que somos capaces de detectar errores en 2 bits y de corregir errores de 1 bits.

   Para poder **detectar los errores** se usan los distintos polinomios que vimos en el ejercicio 1. Como sabemos los bits que son afectados por cada uno de bits de paridad es tan simple como comprobar:

   - Si p_3 XOR x_4 XOR x_3 XOR $x_2 = 1$ detectaría error.
   - Si p_2 XOR x_4 XOR x_3 XOR $x_1 = 1$ detectaría error.
   - Si p_1 XOR x_4 XOR x_2 XOR $x_1 = 1$ detectaría error.

   Asumiendo la cadena de bits como $(x_4x_3x_2p_3x_1p_2p_1)$.

3. **(1.5 puntos) Implemente un programa para Arduino que lea los datos del código de Hamming (7,4) recibidos por el módulo RF PT2272, detecte si hay error, y devuelva por USB a un programa de PC los datos decodificados del código uniforme original. Siga las instrucciones de los seminarios 3 y 4 para realizar este ejercicio.**

   Podemos encontrar este programa en el proyecto adjunto a la práctica. Dentro de la carpeta *cpp*, llamado mainRF2272.cpp.

4. **(1 punto) Explique los fundamentos y el funcionamiento de los códigos CRC. Indique cómo codificar un código de tamaño k=10 a un código CRC de tamaño n=16, según las indicaciones de las clases teóricas y de los seminarios 3 y 4.**

Los códigos de redundancia cíclica (CRC) son un tipo de códigos polinómicos. Un código polinómico está basado en modelar el código con polinomios cuyos coeficientes se expresan en modulo 2.

Utilizan un polinomio generador G(x), el cuál, es un polinomio de orden k utilizado para incluir la redundancia en un CRC. Este polinomio es conocido como el emisor y el receptor.

Se conoce como emisor porque se utiliza para añadir k *bits* de redundancia (detección de errores) a la palabra a transmitir. Estos k *bits* son el resultado del resto de dividir la palabra + k 0's entre el polinomio generador.

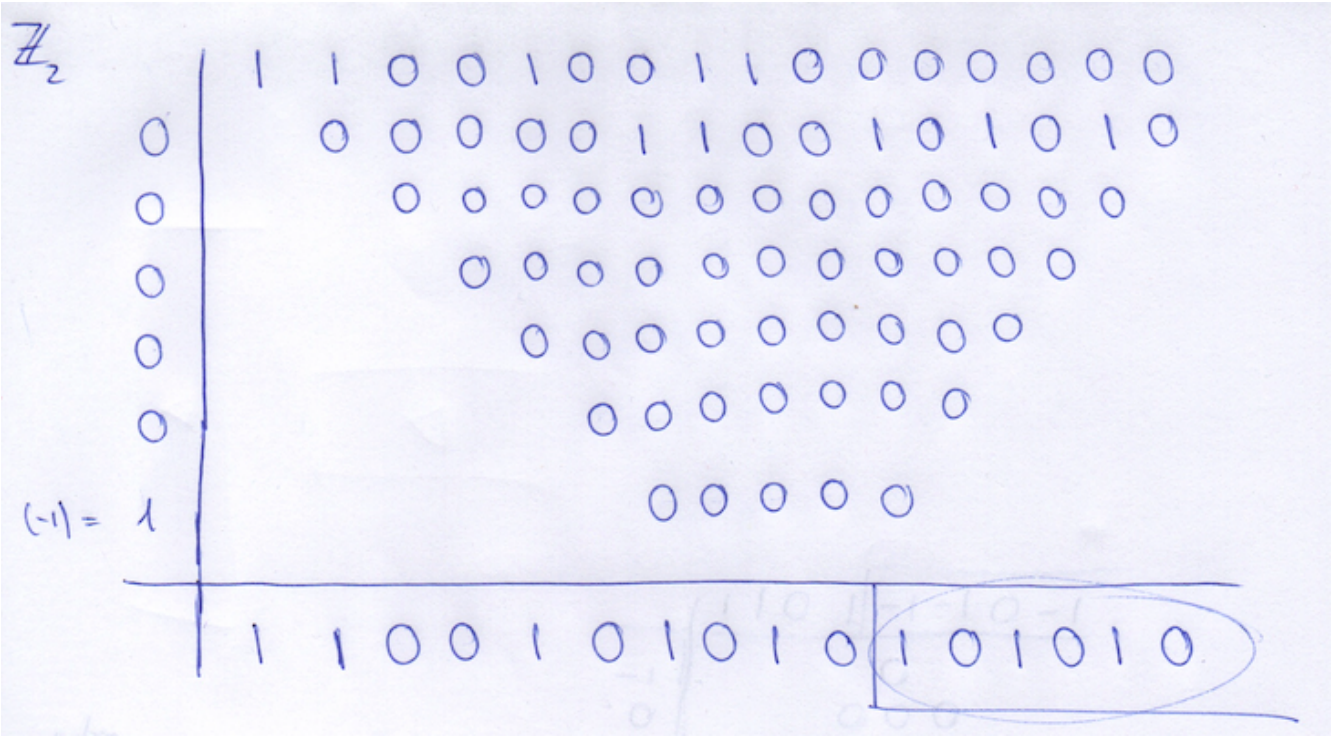
Se conoce como el receptor por que el receptor utiliza el polinomio generador para comprobar si el mensaje recibido tiene o no errores. Unicamente bastaría con comprobar si el mensaje disponible es divisible por el polinomio generador. Siéndolo, el resto es 0 y no hay errores. De otra forma, han ocurrido errores durante la transmisión.

Codificar un código de tamaño k=10 a un codigo CRC de tamaño n=16.

Teniendo un código de tamaño k=10 bits, habría que utilizar un polinomio generador de grado 6 para incluir la redundancia al código original, obteniendo un código CRC de n=16. Por ejemplo:

$$\text{Mensaje perteneciente a código de k=10 bits} \rightarrow m = 1100100110$$
$$\text{Polinomio generador de grado 6} \rightarrow G(x) = x^6 + 1$$

Añadimos 6 ceros al mensaje original (porque el polinomio generador es de grado 6) y lo dividimos entre el polinomio generador, quedándonos con el resto.



Sustituimos los 6 ceros del mensaje por el resto de la división anterior y tendríamos el mensaje con un código CRC de n=16.

Mensaje con código CRC de n=16 $\rightarrow m_{CRC} = 1100100110101010$

5. **(1 punto) Implemente una función en C que tenga como entrada cadenas de 0's y 1's de códigos uniformes de longitud k=5, codifique cada par de códigos con un código CRC-6 (total: 16 bits), según las indicaciones del seminario 4, y devuelva como salida un buffer unsigned char de tamaño máximo 64, con todos los códigos CRC calculados. Rellene con 0's los bits no usados.**

   Tanto el ejercicio 5 como el ejercicio 6 como el 7 no se han pedido implementar en clase por falta de tiempo.

6. **(1 punto) Implemente un programa escritor de tarjetas PICC para Arduino, que tenga como entrada una secuencia de 0's y 1's por USB y que, haciendo uso de la función del ejercicio anterior, codifique la cadena en un buffer y lo guarde en la tarjeta PICC (máximo 64B). Haga uso de las explicaciones de los seminarios 3 y 4 para ello.**

   Tanto el ejercicio 5 como el ejercicio 6 como el 7 no se han pedido implementar en clase por falta de tiempo.

7. **(1 punto) Implemente un programa lector de tarjetas PICC para Arduino, que**

tenga como entrada un buffer de 64B leído desde una tarjeta PICC, compruebe si hay errores en la lectura de cada par de bytes, y vaya guardando en un buffer de salida los códigos uniformes decodificados correctamente, con secuencias de caracteres '0's y '1's. Cuando los 64B hayan sido decodificados correctamente, se deberá enviar la cadena de '0's y '1's por USB mediante puerto serie.

Tanto el ejercicio 5 como el ejercicio 6 como el 7 no se han pedido implementar en clase por falta de tiempo.

8. **(0,5 puntos) Explique cómo plantear un código de Hamming como un código lineal, detallando cómo se construyen las matrices de codificación.**

Un código de Hamming es un tipo particular de un código lineal, por lo que también puede representarse mediante su forma matricial.

Como ejemplo, realizaré la codificación de un código de longitud *k=4* bits con un código de Hamming *(7,4)*.

El mensaje a enviar será x=(1101), el cual ocupa 4 bytes, distribuidos como:

$$\text{x} = (x_4x_3x_2x_1) \rightarrow \text{c} = (x_4x_3x_2p_3x_1p_2p_1)$$

Por tanto la matriz generadora G tendrá un tamaño de k=4 filas y n=7 columnas.

Las posiciones de los bits 3º, 5º, 6º y 7º son los bits del código, por lo que podemos diseñar parcialmente la matriz G:

$$(x_4x_3x_2x_1) \cdot G = (x_4x_3x_2p_3x_1p_2p_1) \rightarrow G = \begin{pmatrix} 1 & 0 & 0 & ? & 0 & ? & ? \\ 0 & 1 & 0 & ? & 0 & ? & ? \\ 0 & 0 & 1 & ? & 0 & ? & ? \\ 0 & 0 & 0 & ? & 1 & ? & ? \end{pmatrix}$$

Para el bit de paridad 3, sabemos que debe tomar paridad par y hemos de tener cuenta los bits desde el 4º hasta el 7º, por lo que $p_3 = x_4 + x_3 + x_2$.

$$(x_4x_3x_2x_1) \cdot G = (x_4x_3x_2p_3x_1p_2p_1) \rightarrow G = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & ? & ? \\ 0 & 1 & 0 & 1 & 0 & ? & ? \\ 0 & 0 & 1 & 1 & 0 & ? & ? \\ 0 & 0 & 0 & 0 & 1 & ? & ? \end{pmatrix}$$

Para el bit de paridad 2, sabemos que debe tomar paridad par y hemos de tener cuenta los bits desde el 2º, 3º, 6º y 7º, por lo que $p_2 = x_4 + x_3 + x_1$.

$$(x_4x_3x_2x_1) \cdot G = (x_4x_3x_2p_3x_1p_2p_1) \rightarrow G = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & ? \\ 0 & 1 & 0 & 1 & 0 & 1 & ? \\ 0 & 0 & 1 & 1 & 0 & 0 & ? \\ 0 & 0 & 0 & 0 & 1 & 1 & ? \end{pmatrix}$$

Para el bit de paridad 1, sabemos que debe tomar paridad par y hemos de tener cuenta los bits desde el 1º, 3º, 5º y 7º, por lo que $p_0 = x_4 + x_2 + x_1$.

$$(x_4x_3x_2x_1) \cdot G = (x_4x_3x_2p_3x_1p_2p_1) \rightarrow G = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Ahora convertimos matriz generadora de Hamming en una matriz generadora de codigo lineal aplicando operaciones de matrices:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} C_1 \leftrightarrow C_3 \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} C_2 \leftrightarrow C_5 \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$C_3 \leftrightarrow C_6 \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} C_2 \leftrightarrow C_7 \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

La matriz quedaría como:

$$\mathrm{M(C)} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Ahora calculamos la matriz $P_{k,n-k}(C)$ a través de M(C):

$$P_{k,n-k}(C) = P_{4,3}(C) = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Seguidamente calculamos H(C) utilizando $[I_{n-k}P^t_{k,n-k}]$:

$$\mathrm{H(C)} = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Como penúltimo paso, calculamos la traspuesta de H(C):

$$H^t(C) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Y para calcular el síndrome solo tendríamos que $s(c) = c \cdot H^t(C)$, donde $c = (c_1c_2c_3c_4c_5c_6c_7) = (1001011)$:

$$s(c) = (1001011) \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = (c_1 + c_4 + c_5 + c_6, c_2 + c_4 + c_6 + c_7, c_3 + c_4 + c_5 + c_7) = (110)$$

9. **(1 punto) Explique cómo calcular el síndrome de un código de Hamming, y cómo se usa el síndrome para conocer en qué bit se encuentra el error.**

   Para el **cálculo del síndrome** necesitamos calcular primero la matriz de control de paridad que es la encargada de obtener el síndrome.

   Esta matriz se calcula fácilmente siguiendo los siguientes pasos:

   - Se incluyen tantas columnas como bits de paridad.
   - Se incluyen tantas filas como longitud del código.
   - Cada columna va asignada a un bit de paridad. Se asignan de izquierda a derecha de menor a mayor.

   Podemos utilizar la imagen que se ve en el ejercicio 1 para saber los valores de esta matriz. Finalmente se obtiene la siguiente matriz:

$$H(C) = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

   Una vez tenemos esta matriz, para calcular el síndrome únicamente tenemos que multiplicar la cadena de bits que queramos comprobar sobre esta matriz:

$$(x_4 x_3 x_2 p_3 x_1 p_2 p_1) \cdot H(C) = \text{síndrome.}$$

El síndrome se utiliza para obtener la posición dentro del vector de bits donde se ha producido el error. Estará compuesto por 3 bits (en este caso) que como conjunto, nos dicen el valor exacto de la posición del error. Por ejemplo, si tenemos un síndrome (101), quiere decir que el error lo tenemos el en bit 5, ya que su codificacion en decimal es 5. Una vez se sabe la posición donde ha ocurrido el error, sólo hay que modificar ese bit y ponerle el valor inverso al que tenga y el error estaría solucionado.

10. **(2 puntos) Implemente un programa para Arduino que lea un código desde el módulo RF PT2272, compruebe si hay errores y los corrija. Finalmente, se deberá enviar el código uniforme (k=4) codificado en el código Hamming por USB a un programa en el PC.**

    Podemos encontrar este programa en el proyecto adjunto a la práctica. Dentro de la carpeta *cpp*, llamado mainRF2272.cpp