Hamming(7,4)

Brayan Steven Mendivelso Pérez
est.brayan.mendive@unimilitar.edu.co
Docente: José de Jesús Rúgeles

***Resumen*— Este laboratorio tiene como objetivo implementar la codificación Hamming (7,4) aplicada a datos del acelerómetro MPU6050, utilizando una Raspberry Pi Pico 2W. Se leen muestras de 16 bits por eje a través de I²C, que se dividen en cuatro nibbles de 4 bits. Cada nibble se codifica con el esquema Hamming (7,4) con paridad par, generando palabras de 7 bits que, concatenadas, forman una muestra de 28 bits. Posteriormente, la información se transmite vía UART y se observa en un osciloscopio. Se proporcionan funciones en Python para realizar la codificación y decodificación, incluyendo la detección y corrección de errores de un solo bit. Finalmente, los estudiantes deben implementar el programa en la Pico 2W, verificar el funcionamiento de la decodificación y analizar los resultados obtenidos, reforzando así el aprendizaje sobre transmisión segura de datos y corrección de errores en sistemas digitales.**

***Abstract*— This laboratory aims to implement Hamming (7,4) coding applied to accelerometer data from the MPU6050 using a Raspberry Pi Pico 2W. Sixteen-bit samples from each axis are read via I²C and divided into four 4-bit nibbles. Each nibble is encoded with the Hamming (7,4) scheme using even parity, generating 7-bit words that, when concatenated, form a 28-bit coded sample. The encoded data is then transmitted through UART and observed on an oscilloscope. Python functions are provided to perform both encoding and decoding, including error detection and single-bit error correction. Students are required to implement the program on the Pico 2W, verify the proper operation of the decoding function, and analyze the obtained results. This practice reinforces knowledge about secure data transmission and error correction in digital systems.**

I. Introducción

La transmisión de datos en sistemas digitales requiere técnicas que garanticen la integridad de la información frente a posibles errores durante la comunicación. Una de las soluciones más empleadas es la codificación Hamming, que permite detectar y corregir errores de un solo bit mediante la inserción de bits de paridad. En este laboratorio se implementa el código Hamming (7,4) utilizando una Raspberry Pi Pico 2W como plataforma de procesamiento. El acelerómetro MPU6050 se emplea únicamente para obtener el dato de aceleración en el eje axa_xax, el cual se toma como muestra de 16 bits y se divide en nibbles de 4 bits para su codificación. Posteriormente, las palabras resultantes se transmiten vía UART y se visualizan en un osciloscopio, lo que permite analizar la trama de datos y comprobar la efectividad de la codificación. De esta manera, se refuerzan conceptos de comunicación digital, transmisión segura y corrección de errores aplicados a sistemas embebidos.dispositivos activos en el bus.

II. Ajuste de código acelerómetro.

Lo primero que se debe realizar en el laboratorio es adaptar el código del acelerómetro MPU6050 para trabajar únicamente con la aceleración en el eje axa_xax. Este valor se convierte a un entero de 16 bits y, posteriormente, se divide en cuatro paquetes de 4 bits (nibbles). Esta separación es necesaria porque cada nibble será utilizado más adelante para aplicar la codificación Hamming (7,4). Para este procedimiento empleamos el siguiente código:

```python
from machine import I2C, Pin
import ssd1306
from imu import MPU6050
from time import sleep

i2c = I2C(0, scl=Pin(5), sda=Pin(4))

oled_ancho = 128
oled_alto = 64
oled = ssd1306.SSD1306_I2C(oled_ancho, oled_alto, i2c)

imu = MPU6050(i2c)
# --- Función para separar en 4 nibbles ---
def dividir_en_nibbles(valor16):
    nibbles = []
    for i in range(4):
        nibble = (valor16 >> (12 - 4*i)) & 0xF   # extrae 4 bits
        nibbles.append(nibble)
    return nibbles

while True:
    # Convertir la aceleración a un entero de 16 bits
    ax = int(imu.accel.x * 1000) & 0xFFFF
    # Separar en 4 nibbles de 4 bits
    nibbles = dividir_en_nibbles(ax)
    # Mostrar en consola (formato binario seguro con f-string)
    print("ax (16 bits):", ax)
    print("Nibbles:", [f"{n:04b}" for n in nibbles])
    # Mostrar en pantalla OLED
    oled.fill(0)
    oled.text("ax:"+str(ax), 0, 0)
    for i, nib in enumerate(nibbles):
        oled.text("N{}: {}".format(i+1, f"{nib:04b}"), 0, 12*(i+1))
    oled.show()
    sleep(0.5)
```

Ilustración 1 adaptación de acelerómetro.

El código desarrollado tiene como objetivo obtener el valor de aceleración en el eje ax del sensor MPU6050, procesarlo en 16 bits y dividirlo en cuatro paquetes de 4 bits. Para ello, primero se configura la comunicación I²C entre la Raspberry Pi Pico 2W y el acelerómetro, además de inicializar la pantalla OLED SSD1306 para la visualización de resultados.

La variable ax almacena la lectura de la aceleración en el eje x, multiplicada por 1000 para aumentar la precisión y posteriormente enmascarada con 0xFFFF para limitarla a 16 bits. Una vez obtenido este valor, la función dividir_en_nibbles() realiza desplazamientos de 4 bits y aplica una máscara 0xF para extraer los cuatro nibbles que conforman el dato original.

Finalmente, el programa presenta el valor completo de ax en consola y en la pantalla OLED, junto con cada nibble en formato binario. De esta forma, se logra separar correctamente la muestra en bloques de 4 bits que serán utilizados en la etapa siguiente de codificación Hamming (7,4).

```
MPY: soft reboot
Unexpected chip ID: 0x98. Possible clone chip?
ax (16 bits): 65397
Nibbles: ['1111', '1111', '0111', '0101']
ax (16 bits): 65397
Nibbles: ['1111', '1111', '0111', '0101']
ax (16 bits): 65395
Nibbles: ['1111', '1111', '0111', '0011']
```

Ilustración 2 resultado de Código.

Al ejecutar el programa, se observa que el valor de aceleración en el eje ax se obtiene como un número entero de 16 bits. Por ejemplo, en una de las lecturas aparece el valor 65397. Al representarlo en binario, este número corresponde a:

$$65397_{10} = 1111\ 1111\ 01111\ 0101_2$$

El código divide este dato en cuatro bloques de 4 bits, llamados nibbles. En este caso, los nibbles resultantes son: 1111, 1111, 0111 y 0101. Esto confirma que la función implementada para separar el valor de 16 bits está funcionando correctamente, ya que cada parte corresponde exactamente a un segmento del número binario original.

Adicionalmente, aparece en consola el mensaje "Unexpected chip ID: 0x98. Possible clone chip?". Este aviso indica que el MPU6050 utilizado podría ser una versión genérica o clonada, pero aun así el sensor responde y permite continuar con la práctica sin inconvenientes.

III. Funciones de codificación y decodificación Hamming(7,4)

Para esta parte el profesor nos suministra el siguiente código

```python
def hamming74_encode(bits4):
    assert len(bits4) == 4 and all(b in (0,1) for b in bits4)
    d3, d2, d1, d0 = bits4
    # Paridades (paridad par)
    P1 = d3 ^ d2 ^ d0             # cubre pos 1,3,5,7
    P2 = d3 ^ d1 ^ d0             # cubre pos 2,3,6,7
    P4 = d2 ^ d1 ^ d0             # cubre pos 4,5,6,7
    # Codeword: [P1, P2, d3, P4, d2, d1, d0]
    return [P1, P2, d3, P4, d2, d1, d0]
def hamming74_decode(code7):
    assert len(code7) == 7 and all(b in (0,1) for b in code7)
    c = code7[:]  # copia para poder corregir
    # Síndrome con paridad par (mismas coberturas que en la cc
    s1 = c[0] ^ c[2] ^ c[4] ^ c[6]   # chequeo P1: pos 1,3,5,7
    s2 = c[1] ^ c[2] ^ c[5] ^ c[6]   # chequeo P2: pos 2,3,6,7
    s4 = c[3] ^ c[4] ^ c[5] ^ c[6]   # chequeo P4: pos 4,5,6,7

    syndrome = s1 + (s2 << 1) + (s4 << 2)   # 0..7 (1-indexadc
    corrected = False
    if syndrome != 0:
        pos = syndrome - 1                  # a índice 0
        c[pos] ^= 1                         # corregir bit
        corrected = True
    # Extraer datos en el orden  [d3,d2,d1,d0]
    data4 = [c[2], c[4], c[5], c[6]]
    return data4, syndrome, corrected, c
```

Ilustración 3 Codificación y decodificación hamming.

En esta etapa del laboratorio se integran las dos partes desarrolladas previamente: el código encargado de obtener la aceleración en el eje axdel sensor MPU6050 y dividirla en cuatro nibbles de 4 bits, junto con las funciones de codificación Hamming (7,4) suministradas por el profesor. Con esta unión, cada uno de los cuatro bloques de 4 bits obtenidos a partir de la muestra de 16 bits se transforma en una palabra de 7 bits mediante la codificación Hamming. De esta forma, la lectura inicial de ax se convierte en un conjunto de 28 bits codificados, listos para ser transmitidos por UART y analizados en el osciloscopio. Esta integración permite observar de manera práctica cómo la información capturada por un sensor se procesa, se segmenta y finalmente se protege frente a errores mediante la aplicación de un código de corrección.

En la implementación inicial de la función hamming74_encode se presentaba un error en el orden de los bits dentro de la palabra codificada. Esto ocasionaba que los resultados de la codificación no fueran compatibles con el proceso de decodificación. Para solucionarlo, se modificó la función de manera que los bits de paridad y de datos quedaran en las posiciones correspondientes al esquema Hamming (7,4) con paridad par.

El nuevo orden de la palabra codificada es:

$$Codeword = [d3, d2, d1, P4, P2, d0, P1]$$

De este modo, los datos [d3,d2,d1,d0]permanecen agrupados al inicio, mientras que los bits de paridad (P1,P2,P4) se colocan en las posiciones finales en el orden inverso. Con este ajuste, se garantiza que la decodificación pueda detectar y corregir errores de un solo bit de manera confiable, cumpliendo el objetivo del laboratorio.

```python
    P2 = d3 ^ d1 ^ d0             # cubre pos 2,3,6,7
    P4 = d2 ^ d1 ^ d0             # cubre pos 4,5,6,7

    # Codeword: [P1, P2, d3, P4, d2, d1, d0]
    return [d3, d2, d1, P4, d0, P2, P1]
```

Ilustración 4 Corrección del código.

Una vez corregido el error en la función de codificación Hamming (7,4), se procedió a probar el sistema completo con los datos del acelerómetro. En este caso, se utilizó exclusivamente la aceleración del eje ax, la cual se obtiene en formato de 16 bits y posteriormente se divide en cuatro nibbles de 4 bits. Cada nibble es enviado a la función de codificación, que lo transforma en una palabra de 7 bits con los bits de paridad correspondientes.

Posteriormente, estas palabras codificadas se sometieron al proceso de decodificación, en el cual se calculan los síndromes y, en caso de encontrar un error de un solo bit, el algoritmo realiza la corrección automáticamente. De esta forma, se comprobó que tanto la codificación como la decodificación funcionan de manera correcta, ya que los datos originales de ax fueron recuperados sin alteraciones después del proceso.

```
ax (16 bits): 65441 hex: 0xffa1
Nib1: 1111 bits:[1, 1, 1, 1] code7:1111111
Nib2: 1111 bits:[1, 1, 1, 1] code7:1111111
Nib3: 1010 bits:[1, 0, 1, 0] code7:1011001
Nib4: 0001 bits:[0, 0, 0, 1] code7:0001111
ax (16 bits): 65444 hex: 0xffa4
Nib1: 1111 bits:[1, 1, 1, 1] code7:1111111
Nib2: 1111 bits:[1, 1, 1, 1] code7:1111111
Nib3: 1010 bits:[1, 0, 1, 0] code7:1011001
Nib4: 0100 bits:[0, 1, 0, 0] code7:0101001
ax (16 bits): 65437 hex: 0xff9d
Nib1: 1111 bits:[1, 1, 1, 1] code7:1111111
Nib2: 1111 bits:[1, 1, 1, 1] code7:1111111
Nib3: 1001 bits:[1, 0, 0, 1] code7:1001100
Nib4: 1101 bits:[1, 1, 0, 1] code7:1100101
ax (16 bits): 65445 hex: 0xffa5
```

Ilustración 5. Codificación y decodificación.

En los resultados obtenidos se observa cómo el valor de la aceleración ax en 16 bits se representa correctamente en formato decimal y hexadecimal, y luego se divide en cuatro nibbles de 4 bits. Por ejemplo, el valor 65444 (0xFFA4) corresponde al número binario 1111 1111 1010 0100, que se separa en los nibbles 1111, 1111, 1010 y 0100. Cada uno de estos bloques de 4 bits fue procesado por la función de codificación Hamming (7,4), generando palabras de 7 bits como 1111111, 1011001 y 0101010, que incluyen tanto la información original como los bits de paridad.

De esta manera, se confirma que el sistema implementado transforma correctamente cada muestra de 16 bits en un conjunto de 28 bits codificados. Además, los resultados muestran coherencia entre los nibbles extraídos y sus correspondientes palabras Hamming, lo que garantiza que el proceso de división y codificación funciona de forma adecuada. Finalmente, este valor codificado fue transmitido mediante la UART y observado en el osciloscopio. Con ello se pudo visualizar la trama correspondiente a los 28 bits resultantes, lo que permitió comprobar en tiempo real el comportamiento de la señal transmitida y validar la correcta implementación del proceso de codificación.

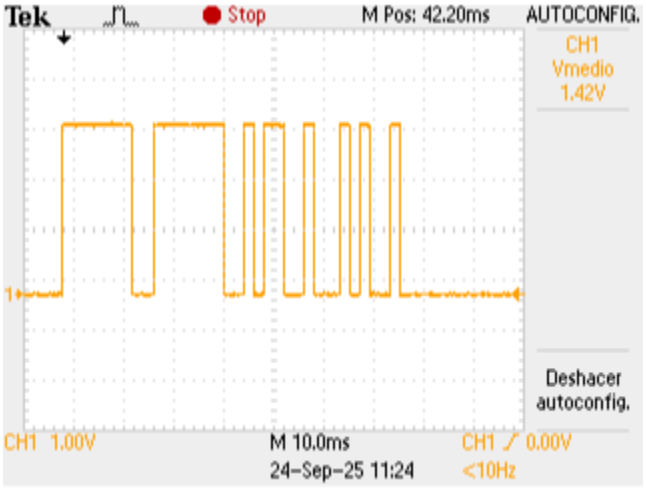

Ilustración 6 Señal en osciloscopio.

En la figura se presenta la señal capturada en el osciloscopio correspondiente a la transmisión por UART del valor de la aceleración ax=65444. Este valor fue previamente dividido en cuatro nibbles de 4 bits y luego codificado utilizando el esquema Hamming (7,4), generando un total de 28 bits de información protegida frente a errores. La trama completa es enviada de forma serial por el pin Tx de la Raspberry Pi Pico 2W y visualizada en el osciloscopio digital.

La señal obtenida corresponde a una secuencia de pulsos digitales que representan los valores binarios transmitidos. Se pueden identificar claramente los niveles lógicos: un nivel alto, que representa el bit lógico “1”, y un nivel bajo, que corresponde al bit lógico “0”. Asimismo, en la trama observada es posible distinguir el bit de inicio (start bit), los bits de datos codificados y el bit de parada (stop bit), que forman parte del protocolo UART.

La forma de onda refleja el comportamiento típico de una transmisión serial asincrónica: cada grupo de pulsos corresponde a un byte transmitido, dentro del cual se encapsulan las palabras de 7 bits generadas por el código Hamming (7,4). En este caso, los cuatro nibbles del dato original (65444) fueron transformados en cuatro palabras de 7 bits, concatenadas y enviadas una tras otra a través del canal de comunicación.

La visualización en el osciloscopio confirma que la implementación es correcta, ya que la señal transmitida coincide con lo esperado para un tren de pulsos binarios provenientes de la codificación. Además, el análisis temporal de la señal permite comprobar la velocidad de transmisión configurada y verificar la estabilidad de los niveles lógicos. Este resultado demuestra que el sistema no solo procesa y codifica los datos de manera adecuada, sino que también los transmite de forma fiable, siendo posible observar en tiempo real la trama de 28 bits codificada según el esquema Hamming (7,4).

IV. Análisis.

El laboratorio desarrollado tuvo como propósito principal aplicar el código Hamming (7,4) en un sistema real de adquisición y transmisión de datos, integrando teoría de corrección de errores con práctica en sistemas embebidos. La actividad se abordó en varias etapas, cada una de las cuales permitió afianzar conceptos fundamentales de comunicación digital y programación en microcontroladores.

En la primera fase, se configuró la comunicación I²C entre la Raspberry Pi Pico 2W y el sensor MPU6050, seleccionando únicamente el eje de aceleración ax como fuente de datos. Este valor, representado en 16 bits, fue procesado para convertirlo en un entero y adaptarlo al sistema de codificación. La decisión de utilizar un solo eje simplificó el análisis sin perder la generalidad del procedimiento, ya que el mismo método puede aplicarse a los demás ejes del sensor.

Posteriormente, se implementó un algoritmo que divide el valor de 16 bits en cuatro nibbles de 4 bits. Este paso es fundamental porque el código Hamming (7,4) está diseñado para trabajar con bloques de 4 bits de datos, a los que se les agregan 3 bits de paridad, formando así palabras de 7 bits. De este modo, una sola muestra del acelerómetro genera cuatro palabras codificadas, resultando en un paquete de 28 bits.

En la segunda fase se trabajó con las funciones de codificación y decodificación. El código inicial suministrado por el profesor contenía un error en el orden de los bits de salida, lo que ocasionaba inconsistencias en la decodificación. Fue necesario corregir el algoritmo de hamming74_encode para garantizar que los bits de paridad se ubicaran correctamente y que la palabra final siguiera la estructura estándar. Esta modificación permitió que el proceso de decodificación funcionara de manera adecuada, detectando errores mediante el síndrome y corrigiendo un solo bit afectado en la transmisión.

Una vez corregido el código, se realizaron pruebas prácticas en consola, mostrando tanto los nibbles obtenidos como las palabras de 7 bits generadas por la codificación. Por ejemplo, el valor =65444 (0xFFA4) fue transformado en los nibbles 1111, 1111, 1010 y 0100. Cada uno fue codificado como 1111111, 1111111, 1011001 y 0101010, confirmando que la codificación se realizó de acuerdo con la teoría del código Hamming. Estos resultados demostraron que de un dato inicial de 16 bits se obtuvo un paquete de 28 bits protegido frente a errores.

La última etapa consistió en la transmisión de los datos por UART y su visualización en un osciloscopio digital. El tren de pulsos observado correspondió exactamente a la secuencia binaria generada por la codificación, donde fue posible distinguir los niveles lógicos altos y bajos, así como los bits de inicio y parada característicos del protocolo UART. El análisis temporal de la señal permitió además validar la tasa de transmisión configurada y la estabilidad del canal de comunicación.

En conjunto, el laboratorio integró la teoría de codificación de canal con la práctica en hardware embebido. Se logró comprender no solo cómo se aplican los códigos de Hamming a bloques de información, sino también cómo se transmiten y visualizan físicamente en un osciloscopio, cerrando el ciclo

completo desde la adquisición de datos hasta la comunicación segura.

V. Conclusiones.

El desarrollo de este laboratorio permitió comprender de forma integral cómo se aplican los códigos de detección y corrección de errores en un entorno práctico con microcontroladores. El trabajo inició con la configuración de la comunicación I²C entre la Raspberry Pi Pico 2W y el sensor MPU6050, tomando únicamente el eje axa_xax como fuente de datos. Este valor, representado en 16 bits, fue transformado en un entero y posteriormente dividido en cuatro nibbles de 4 bits, lo cual facilitó la aplicación del código Hamming (7,4). La segmentación resultó fundamental, ya que evidenció cómo los datos obtenidos de un sensor deben ser adaptados al tamaño de bloque requerido por el algoritmo de codificación. En la etapa de implementación se identificó un error en el código original de Hamming suministrado, el cual generaba resultados incorrectos en la ubicación de los bits de paridad; este fallo fue corregido y, con ello, se garantizó que las palabras de 7 bits se construyeran de forma adecuada y que el proceso de decodificación funcionara de manera confiable, permitiendo recuperar los datos originales incluso frente a errores de un solo bit. Las pruebas realizadas en consola confirmaron que cada nibble era transformado en una palabra Hamming coherente y consistente con la teoría, mientras que la transmisión de las tramas por UART y su posterior visualización en el osciloscopio ofrecieron una validación física del proceso: en la señal se observaron claramente los niveles lógicos altos y bajos, así como los bits de inicio, de datos y de parada propios del protocolo serial. Con este flujo completo, desde la adquisición de datos hasta la transmisión y análisis en el osciloscopio, se logró integrar conceptos de programación, procesamiento digital de señales y comunicaciones, reforzando la importancia de aplicar técnicas de corrección de errores en la práctica. En conclusión, el laboratorio no solo consolidó los conocimientos teóricos sobre el código Hamming (7,4), sino que también mostró su relevancia en la construcción de sistemas robustos de comunicación digital, donde garantizar la integridad de los datos es un aspecto esencial tanto en sistemas embebidos como en aplicaciones de telecomunicaciones modernas.

Referencias

[1] R. W. Hamming, “Error detecting and error correcting codes,” *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, Apr. 1950.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2014.

[3] Raspberry Pi Foundation, *Raspberry Pi Pico Python SDK Documentation*, 2022. [En línea]. Disponible en: https://www.raspberrypi.com/documentation/microcontrollers/ [Accedido: 24-sep-2025].

[4] InvenSense, *MPU-6000 and MPU-6050 Product Specification Revision 3.4*, 2013. [En línea]. Disponible en: https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6050/ [Accedido: 24-sep-2025].

[5] J. F. Wakerly, *Digital Design: Principles and Practices*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall, 2006.