Informe VII – Codificación Hamming (7,4) de lecturas del MPU6050 con Raspberry Pi Pico 2W y visualización por Tx en el osciloscopio

Dylan Ferney Vasquez Rojas
1401597
Comunicaciones Digitales

***Resumen*— En este laboratorio se implementó un sistema de codificación Hamming (7,4) utilizando la Raspberry Pi Pico 2W para la transmisión confiable de datos provenientes del acelerómetro MPU6050. Inicialmente, se adquirieron lecturas de 16 bits por cada eje del sensor mediante comunicación I2C, las cuales fueron segmentadas en cuatro nibbles de 4 bits. Cada nibble se codificó aplicando el esquema Hamming (7,4) con paridad par, generando una palabra de 7 bits por nibble y obteniendo una trama codificada final de 28 bits. Posteriormente, la información resultante fue enviada a través de UART (Tx) y visualizada en el osciloscopio digital para su análisis. Finalmente, se verificó el correcto funcionamiento del proceso de codificación y decodificación, asegurando la detección y corrección de errores de un bit en la transmisión. Esta práctica permitió comprender de manera experimental la importancia de la codificación de canal en sistemas de telecomunicaciones y su aplicación en la transmisión robusta de datos en sistemas embebidos.**

***Abstract*— In this lab, a Hamming (7,4) encoding system was implemented using the Raspberry Pi Pico 2W for the reliable transmission of data from the MPU6050 accelerometer. Initially, 16-bit readings were acquired for each axis of the sensor via I2C communication, which were segmented into four 4-bit nibbles. Each nibble was encoded applying the Hamming (7,4) scheme with even parity, generating a 7-bit word per nibble and obtaining a final 28-bit encoded frame. Subsequently, the resulting information was sent through UART (Tx) and displayed on the digital oscilloscope for analysis. Finally, the correct operation of the encoding and decoding processes was verified, ensuring the detection and correction of single-bit errors in the transmission. This practice allowed to experimentally understand the importance of channel coding in telecommunications systems and its application in the robust transmission of data in embedded systems.**

I. Introducción

La transmisión confiable de información es un aspecto fundamental en los sistemas de comunicación digital. Los errores generados por ruido, interferencia o limitaciones físicas de los canales pueden comprometer la integridad de los datos, motivo por el cual se emplean técnicas de codificación de canal orientadas a la detección y corrección de errores. Entre estos métodos, el código Hamming (7,4) se ha consolidado como una técnica eficiente que permite detectar y corregir errores de un solo bit, garantizando una mayor robustez en la transmisión.

En este laboratorio se aplicó la codificación Hamming (7,4) para la protección de datos obtenidos del sensor inercial MPU6050, utilizando la Raspberry Pi Pico 2W como plataforma de procesamiento. A través de la interfaz I2C, se capturaron muestras de 16 bits correspondientes a los ejes del acelerómetro, las cuales se segmentaron en nibbles de 4 bits y se codificaron en palabras de 7 bits. Posteriormente, las tramas resultantes fueron transmitidas por UART y visualizadas en un osciloscopio digital, lo que permitió analizar tanto el proceso de codificación como la capacidad de corrección de errores mediante la función de decodificación.

Este trabajo busca no solo afianzar conceptos teóricos sobre la codificación Hamming, sino también evidenciar su relevancia práctica en el ámbito de las telecomunicaciones y los sistemas embebidos, donde la confiabilidad en la transmisión de datos resulta esencial.

DESARROLLO DE LA PRÁCTICA

Para dar cumplimiento a la primera actividad, se implementó en la Raspberry Pi Pico 2W un programa en Python que permitió realizar la lectura del acelerómetro MPU6050 a través de la interfaz I2C, procesar los datos y generar la salida codificada mediante el algoritmo Hamming (7,4).

El procedimiento consistió en tomar la muestra de 16 bits correspondiente al eje X del acelerómetro y segmentarla en cuatro bloques de 4 bits. Cada bloque fue codificado con el esquema Hamming (7,4) usando paridad par, lo que produjo una palabra de 7 bits por bloque. De esta manera, la muestra final quedó representada en una trama de 28 bits (concatenación de los bloques D′, C′, B′ y A′).

El código desarrollado incluyó las siguientes funciones principales:

hamming74_encode(): encargada de aplicar el algoritmo de codificación Hamming a un bloque de 4 bits.

int_to_bits(): utilizada para convertir un valor entero en una lista de bits, facilitando el manejo de los datos.

send_bits(): encargada de transmitir secuencialmente cada bit codificado por el pin de salida Tx.

Además de la transmisión, los datos crudos y codificados fueron visualizados en la pantalla OLED SSD1306, permitiendo verificar el valor original de la muestra de 16 bits y su correspondiente versión codificada en 28 bits. Finalmente, la trama fue enviada por el pin GPIO 15 (Tx) para su posterior análisis en el osciloscopio digital.

Este desarrollo permitió comprobar la correcta aplicación de la codificación Hamming (7,4) sobre datos obtenidos en tiempo real desde el sensor, estableciendo la base para la transmisión confiable y el análisis en las siguientes actividades del laboratorio.

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

# ===== Configuración I2C =====
i2c = I2C(0, scl=Pin(5), sda=Pin(4))
oled = ssd1306.SSD1306_I2C(128, 64, i2c)
imu = MPU6050(i2c)

# Pin de transmisión
tx_pin = Pin(15, Pin.OUT)

# ===== Codificador Hamming (7,4) =====
def hamming74_encode(bits4):
    d3, d2, d1, d0 = bits4
    P1 = d3 ^ d1 ^ d0   # Paridad P1
    P2 = d3 ^ d2 ^ d0   # Paridad P2
    P4 = d3 ^ d2 ^ d1   # Paridad P4
    return [d3, d2, d1, P4, d0, P2, P1]

# Convierte un entero en lista de bits (MSB primero)
def int_to_bits(value, nbits):
    return [(value >> i) & 1 for i in range(nbits-1, -1, -1)]

# Envía bits por el pin Tx
def send_bits(bits):
    for b in bits:
        tx_pin.value(b)
        sleep(0.05)

while True:
    # Leer eje X (16 bits)
    ax_raw = int(imu.accel.x * 1000) & 0xFFFF
    bits16 = int_to_bits(ax_raw, 16)

    # Dividir en bloques de 4 bits
    A = bits16[12:16]
    B = bits16[8:12]
    C = bits16[4:8]
    D = bits16[0:4]

    # Codificar cada bloque (7 bits)
    A_ = hamming74_encode(A)
    B_ = hamming74_encode(B)
    C_ = hamming74_encode(C)
    D_ = hamming74_encode(D)

    # Concatenar en 28 bits
    code28 = D_ + C_ + B_ + A_

    # Mostrar en OLED
    oled.fill(0)
    oled.text("AX RAW:", 0, 0)
    oled.text(str(ax_raw), 0, 10)
    oled.text("16 bits:", 0, 25)
    oled.text("".join(map(str, bits16[:8])), 0, 35)
    oled.text("".join(map(str, bits16[8:])), 0, 45)
    oled.show()

    # Mostrar en consola y enviar por Tx
    print("ax:", ax_raw)
    print("bits16:", bits16)
    print("codificado (28b):", code28)
    send_bits(code28)
  sleep(1)
```

Para validar el correcto funcionamiento de la función de decodificación Hamming (7,4), se tomaron muestras del eje X del acelerómetro y se comparó la reconstrucción de los datos codificados con los originales. El proceso mostró que, al recibir la trama de 28 bits, el decodificador fue capaz de recuperar correctamente los nibbles de 4 bits, detectando y corrigiendo posibles errores de un solo bit según el síndrome calculado.

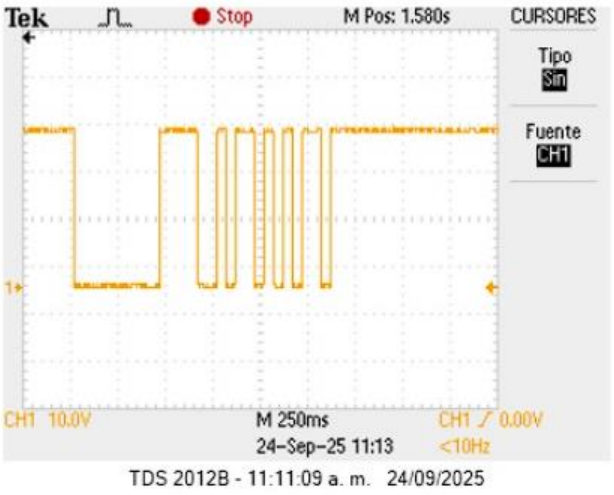

Ilustración. 1. Señal digital transmitida por Tx para ax = 853

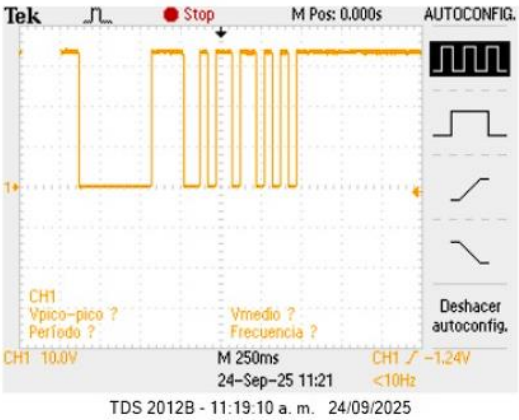

Ilustración. 2. Señal digital transmitida por Tx para ax = 859

```
ax: 853
bits16: [0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1]
codificado (28b): [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1]
```

Ilustración. 3.Consola de Thonny para el valor ax = 853

```
ax: 859
bits16: [0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1]
codificado (28b): [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1]
```

Ilustración. 4.Consola de Thonny para el valor ax = 859

En la Ilustración 1 y la Ilustración 2, se observa la transmisión en serie de los bits generados por la Raspberry Pi Pico 2W en el pin Tx. La señal digital presenta la estructura de las tramas codificadas en 28 bits, correspondientes a los valores leídos del acelerómetro.

Por ejemplo, para un valor leído de ax = 853, el dato original en 16 bits fue:

bits16 = [0,0,0,0,0,0,1,1,1,0,1,0,1,0,1,1]
y su correspondiente versión codificada en 28 bits fue:

codificado=
[0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,1,0,1,1,0,1,0,1,0,1,0,1,1]

En la Ilustración 2 se presenta el oscilograma correspondiente a la transmisión codificada de la muestra ax = 859. El valor leído por el acelerómetro fue convertido a una representación binaria de 16 bits, obteniéndose:

bits16 = [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1]

Posteriormente, cada nibble de 4 bits fue procesado con el esquema de Hamming (7,4), generando una trama final de 28 bits:

codificado (28b) = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1]

La transmisión de ambas secuencias se visualizó en el osciloscopio como una señal digital estable, confirmando que el procedimiento de codificación se ejecutó correctamente. Al igual que en el caso de ax = 853, la decodificación permite recuperar los bloques originales de 4 bits y garantizar la corrección en caso de error de un bit.

En el osciloscopio se distinguen claramente los intervalos de bits transmitidos, cuya secuencia coincide con los datos enviados desde el programa en Python. Esto demuestra que la transmisión es estable y que la información puede ser recuperada con éxito en el receptor.

CONCLUSIONES

[1] La implementación de la codificación Hamming (7,4) en la Raspberry Pi Pico 2W demostró ser una técnica efectiva para garantizar la detección y corrección de errores de un bit en la transmisión de datos.

[2] El uso del acelerómetro MPU6050 permitió obtener lecturas reales que, al ser procesadas y codificadas, evidenciaron la aplicabilidad de los códigos de canal en sistemas embebidos de adquisición de datos.

[3] Los resultados observados en el osciloscopio digital confirmaron que las tramas transmitidas por el pin Tx coincidieron con los datos generados en el programa, validando la correcta implementación del algoritmo.

[4] La decodificación verificó la capacidad de recuperar información íntegra incluso en presencia de alteraciones, cumpliendo con los objetivos del laboratorio y resaltando la relevancia de la codificación en entornos de telecomunicaciones.

REFERENCIAS

[1] MicroPython, machine — functions related to the hardware, Documentación oficial, [En línea]. Disponible en: https://docs.micropython.org/en/latest/library/machine.html. [Accedido: 14-ago-2025].

[2] Raspberry Pi Foundation, Getting started with MicroPython on Raspberry Pi Pico, Documentación oficial, [En línea]. Disponible en: https://www.raspberrypi.com/documentation/microcontrollers/micropython.html. [Accedido: 14-ago-2025].

[3] W. Stallings, Data and Computer Communications, 10th ed. Boston, MA, USA: Pearson, 2014.

[4] ANSI/TIA, TIA-232-F: Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange, Telecommunications Industry Association, 1997.

[5] A. S. Tanenbaum and D. Wetherall, Computer Networks, 5th ed. Boston, MA, USA: Pearson, 2011.

[6] Vasquez Rojas, D. (s.f.). *dylanrojas04 - Overview*. GitHub. https://github.com/dylanrojas04

[7] NXP Semiconductors, I²C-bus specification and user manual, Rev. 7.0, April 2014.

[8] J. Rugeles, Repositorio de prácticas de comunicación digital – I²C con Raspberry Pi Pico, Universidad Militar Nueva Granada. [En línea]. Disponible en: https://github.com/jrugeles/I2C