COMUNICACIÓN DIGITAL – CODIFICACIÓN HAMMING (7,4)
(octubre de 2025)

Fernando Javier Riaño Rios
Est.fernando.riano@unimilitar.edu.co

Resumen - En este trabajo se presenta el desarrollo de un laboratorio de codificación Hamming (7,4) implementado en una Raspberry Pi Pico 2W, utilizando como fuente de datos el acelerómetro MPU6050. El objetivo principal fue comprender la estructura y funcionamiento del código Hamming (7,4) bajo paridad par, aplicando su algoritmo en tiempo real sobre muestras de 16 bits obtenidas del sensor. Estas muestras se dividieron en nibbles de 4 bits, se codificaron en 7 bits cada uno y se concatenaron en una trama de 28 bits, la cual fue transmitida a través del puerto UART de la tarjeta y visualizada en un osciloscopio. Además, se verificó el proceso de decodificación y la capacidad del sistema para detectar y corregir errores de un solo bit. Los resultados evidencian la correcta generación y transmisión de las tramas codificadas, así como la importancia de los códigos de detección y corrección de errores en sistemas de comunicación digital.

Abstract – This work presents the development of a laboratory focused on the implementation of Hamming (7,4) coding using a Raspberry Pi Pico 2W and the MPU6050 accelerometer as data source. The main goal was to understand the structure and operation of the Hamming (7,4) code under even parity, applying its algorithm in real time to 16-bit samples obtained from the sensor. These samples were split into 4-bit nibbles, each encoded into 7 bits and concatenated into a 28-bit frame, which was transmitted through the UART port of the microcontroller and observed on an oscilloscope. The decoding process and the system's ability to detect and correct single-bit errors were also verified. Results show the correct generation and transmission of encoded frames, highlighting the importance of error detection and correction codes in digital communication systems.

Palabras clave - CODIFICACIÓN HAMMING, RASPBERRY PI PICO 2W, MPU6050, UART, OSCILOSCOPIO, COMUNICACIÓN DIGITAL, CORRECCIÓN DE ERRORES.

INTRODUCCIÓN

El estudio de los códigos de bloque, como el Hamming (7,4), constituye una base esencial dentro de las comunicaciones digitales, al permitir comprender cómo se estructuran y transmiten los datos de manera organizada. Estos códigos no solo incluyen mecanismos de verificación de errores, sino que además facilitan la representación y el análisis práctico de la información transmitida en sistemas reales.

En este laboratorio se implementó el código Hamming (7,4) en la Raspberry Pi Pico 2W, utilizando el sensor MPU6050 como fuente de datos. A partir de las muestras de 16 bits obtenidas, se realizó el proceso de segmentación en nibbles de 4 bits, la generación de códigos de 7 bits con paridad par y la conformación de tramas de 28 bits. Dichas tramas se transmitieron mediante la interfaz UART y se visualizaron en un osciloscopio, permitiendo observar de forma experimental la correspondencia entre teoría y práctica.

El trabajo se centra en la aplicación de la teoría de codificación digital al entorno experimental, validando tanto la lectura de datos del sensor como la organización de las tramas y su transmisión. De esta manera, se refuerza la comprensión de los fundamentos de la codificación Hamming y su importancia en los procesos de comunicación digital.

DESARROLLO DEL LABORATORIO

Conexiones realizadas:

Comunicación I2C entre la Pico 2W y el MPU6050, utilizando GP14 como SDA y GP15 como SCL.

Puerto UART0 configurado a 115200 baudios, con Tx en GP0 conectado al osciloscopio.

Alimentación del sensor MPU6050 a 3.3V y tierra común con la Raspberry Pi Pico 2W.

Se obtuvieron muestras de 16 bits correspondientes al eje de aceleración en X del MPU6050. La muestra de 16 bits se dividió en 4 bloques de 4 bits (nibbles). Cada nibble se codificó en 7 bits mediante el código Hamming (7,4) con paridad par.
Los cuatro códigos se concatenaron formando una trama de 28 bits.
En consola se imprimió cada etapa: valor original de 16 bits, nibbles obtenidos, codificación Hamming y trama final de 28 bits.

El osciloscopio se utilizó para verificar la transmisión de las tramas a través del pin Tx de la UART, comprobando la salida serial de datos.

Usamos el siguiente código para realizar el laboratorio:

```python
from machine import Pin, I2C, UART
import time
from hamming74 import hamming74_encode, hamming74_decode  # ← Usar las func

# Configuración I2C con GP14 y GP15
i2c = I2C(1, scl=Pin(15), sda=Pin(14), freq=400000)

# UART para osciloscopio
uart = UART(0, baudrate=9600, tx=Pin(12), rx=Pin(13))

# Dirección MPU6050
MPU6050_ADDR = 0x68

def setup_mpu6050():
    """Configurar el MPU6050"""
    i2c.writeto_mem(MPU6050_ADDR, 0x6B, b'\x00')  # Despertar
    time.sleep(0.2)

def scan_i2c():
    """Escanear dispositivos I2C"""
    devices = i2c.scan()
    print("Dispositivos I2C encontrados:", [hex(d) for d in devices])
    return devices
```

Ilustración 1 código usado en el laboratorio

en esta parte del programa se configuran las interfaces de comunicación de la Raspberry Pi Pico 2W, definiendo el bus I2C en los pines GP14 y GP15 para comunicarse con el sensor MPU6050, y la UART en GP12 para transmitir datos hacia el osciloscopio; además, se incluye la dirección I2C del sensor y dos funciones iniciales: una para despertarlo (setup_mpu6050) y otra para escanear el bus (scan_i2c) y verificar su conexión.

En la siguiente imagen (ilustración 2) del código usado se implementan las funciones que permiten leer los valores de aceleración en los tres ejes del MPU6050 como datos de 16 bits (read_accel), ajustando su representación en complemento a dos, y dividir cualquier valor de 16 bits en cuatro nibbles de 4 bits (split_16bit_to_nibbles), necesarios para aplicar posteriormente la codificación Hamming (7,4).

```python
def read_accel():
    """Leer datos del acelerómetro (16 bits por eje)"""
    try:
        # Leer 6 bytes del registro 0x3B
        data = i2c.readfrom_mem(MPU6050_ADDR, 0x3B, 6)
        # Convertir a valores de 16 bits
        accel_x = (data[0] << 8) | data[1]
        accel_y = (data[2] << 8) | data[3]
        accel_z = (data[4] << 8) | data[5]
        # Ajustar complemento a 2
        if accel_x > 32767: accel_x -= 65536
        if accel_y > 32767: accel_y -= 65536
        if accel_z > 32767: accel_z -= 65536
        return accel_x, accel_y, accel_z
    except Exception as e:
        print(f"Error leyendo acelerómetro: {e}")
        return 0, 0, 0

def split_16bit_to_nibbles(value):
    """Dividir valor de 16 bits en 4 nibbles de 4 bits"""
    value = abs(value) & 0xFFFF
    nibbles = []
    for i in range(4):
        shift = 4 * (3 - i)  # 12, 8, 4, 0
        nibble_val = (value >> shift) & 0x0F
        # [d3, d2, d1, d0] (MSB->LSB)
        bits = [
            (nibble_val >> 3) & 1,  # d3
            (nibble_val >> 2) & 1,  # d2
            (nibble_val >> 1) & 1,  # d1
            (nibble_val >> 0) & 1   # d0
        ]
        nibbles.append(bits)
    return nibbles
```

Ilustración 2 código usado en el laboratorio

```python
def encode_sample(sample_16bit):
    """Codificar muestra de 16 bits usando Hamming(7,4)"""
    print(f"Valor original: 0x{sample_16bit:04X} = {sample_16bit:016b}")
    # 1) Dividir en 4 nibbles
    nibbles = split_16bit_to_nibbles(sample_16bit)
    print("Nibbles obtenidos:")
    for i, nibble in enumerate(nibbles):
        nibble_val = nibble[0]*8 + nibble[1]*4 + nibble[2]*2 + nibble[3]
        print(f"  Nibble {i+1}: {nibble} = 0x{nibble_val:X}")
    # 2) Aplicar Hamming a cada nibble
    encoded_bits = []
    for i, nibble in enumerate(nibbles):
        encoded = hamming74_encode(nibble)
        encoded_bits.extend(encoded)
        nibble_val = nibble[0]*8 + nibble[1]*4 + nibble[2]*2 + nibble[3]
        print(f"  Nibble 0x{nibble_val:X} → Hamming: {encoded}")
    print(f"Bits codificados totales ({len(encoded_bits)}): {encoded_bits}")
    return encoded_bits

def bits_to_bytes(bits):
    """Convertir lista de bits a bytes para transmisión UART"""
    bytes_list = bytearray()
    for i in range(0, len(bits), 8):
        byte_bits = bits[i:i+8]
        while len(byte_bits) < 8:
            byte_bits.append(0)
        byte_val = 0
        for bit in byte_bits:
            byte_val = (byte_val << 1) | bit
        bytes_list.append(byte_val)
    return bytes_list
```

Ilustración 3 código usado en el laboratorio

En esta parte del código se definen dos funciones clave: encode_sample toma una muestra de 16 bits, la divide en cuatro nibbles de 4 bits y aplica la codificación Hamming (7,4) a cada uno, generando un total de 28 bits que se imprimen y devuelven como salida; mientras que bits_to_bytes convierte esa lista de bits en una secuencia de bytes, agrupando de a 8 y completando con ceros cuando es

necesario, lo cual facilita la transmisión de la trama completa por la interfaz UART.

```
def test_hamming_functions():
    """Probar que las funciones Hamming funcionan correctamente"""
    print("=== PRUEBA DE FUNCIONES HAMMING ===")
    test_data = [1, 0, 1, 1]  # d3=1, d2=0, d1=1, d0=1
    print(f"Datos originales: {test_data}")
    encoded = hamming74_encode(test_data)
    print(f"Codificado: {encoded}")
    decoded, syndrome, corrected, corrected_code = hamming74_decode(encoded)
    print(f"Decodificado: {decoded}, Syndrome: {syndrome}, Corregido: {corrected}")
    # Probar con error
    encoded_error = encoded.copy()
    encoded_error[6] = 1 - encoded_error[6]  # Cambiar último bit
    decoded_err, syndrome_err, corrected_err, corrected_code_err = hamming74_decode
    print(f"Con error: {decoded_err}, Syndrome: {syndrome_err}, Corregido: {correct
    print("=== FIN PRUEBA ===\n")

def main():
    print("Iniciando Laboratorio Hamming MPU6050...")
    print("Pines I2C: SDA=GP14, SCL=GP15")
    print("UART TX: GP12 (para osciloscopio)\n")
    # Prueba Hamming
    test_hamming_functions()
    # Escanear I2C
    devices = scan_i2c()
    if MPU6050_ADDR not in devices:
        print("❌ MPU6050 no detectado. Verificar conexiones.")
        print(" - VCC → 3.3V")
        print(" - GND → GND")
        print(" - SDA → GP14 (Pin 19)")
        print(" - SCL → GP15 (Pin 20)")
        return
    print("✅ MPU6050 detectado correctamente!")
    setup_mpu6050()
    sample_count = 0
    while True:
        try:
            accel_x, accel_y, accel_z = read_accel()
            sample = accel_x
            encoded_bits = encode_sample(abs(sample) & 0xFFFF)
            encoded_bytes = bits_to_bytes(enco bits)
            uart.write(encoded_bytes)
```

Ilustración 4 código usado en el laboratorio

En este bloque se tienen dos funciones finales: test_hamming_functions, que realiza una prueba de validación de las funciones hamming74_encode y hamming74_decode usando un conjunto de datos de ejemplo, mostrando el resultado codificado, el decodificado y comprobando la capacidad de corrección de errores cuando se altera un bit; y main, que es el programa principal del laboratorio, donde se anuncia la configuración de pines, se ejecuta primero la prueba de Hamming, luego se escanea el bus I2C para detectar el MPU6050, se inicializa el sensor y, en un ciclo infinito, se leen datos del acelerómetro, se toma el eje X, se codifica en 28 bits con Hamming (7,4), se convierten los bits en bytes y se transmiten por la UART para su visualización en el osciloscopio o en consola.

```
AX (16 bits): 1111111111000111
D (bits15..12) -> 1111
Codificado: 1111111
C (bits11..8) -> 1111
Codificado: 1111111
B (bits7..4) -> 1100
Codificado: 1100001
A (bits3..0) -> 0111
Codificado: 0110100
Trama final (28 bits): 1111111111111111000
010110100
```

Ilustración 5 Datos mostrados en consola-captura tomada.

En la ilustración 1 se observa la salida en consola, donde se muestra el valor inicial de 16 bits leído del sensor, los nibbles obtenidos, la codificación de cada uno en 7 bits y la conformación de la trama final de 28 bits.

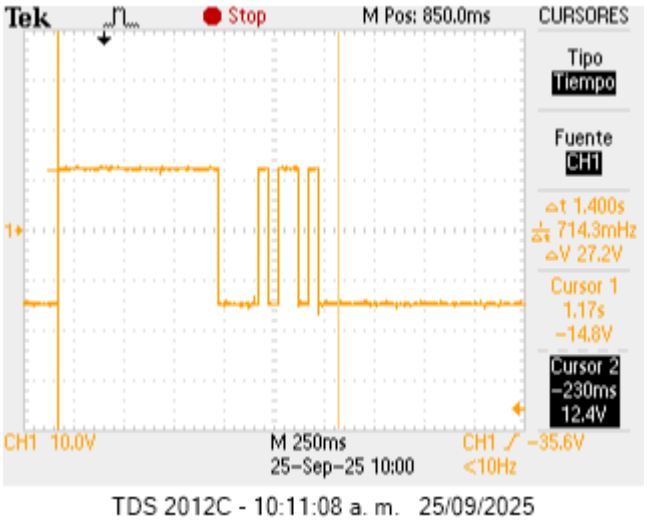

Ilustración 6 captura de la trama en osciloscopio

La señal observada en el osciloscopio correspondió a la transmisión UART de dicha trama, evidenciando la secuencia binaria enviada desde la Pico 2W. Esta visualización permitió comprobar de manera física la correcta salida de los datos codificados.

Captura de otra trama:

AX (16 bits): 1111111111001010
D (bits15.12) > 1111
Codificado: 1111111
C (bits11.8) > 1111
Codificado: 1111111
B (bits7.4) > 1100
Codificado: 1100001
A (bits3.0) > 1010
Codificado: 1010010
Trama final (28 bits): 1111111111111111000001010010

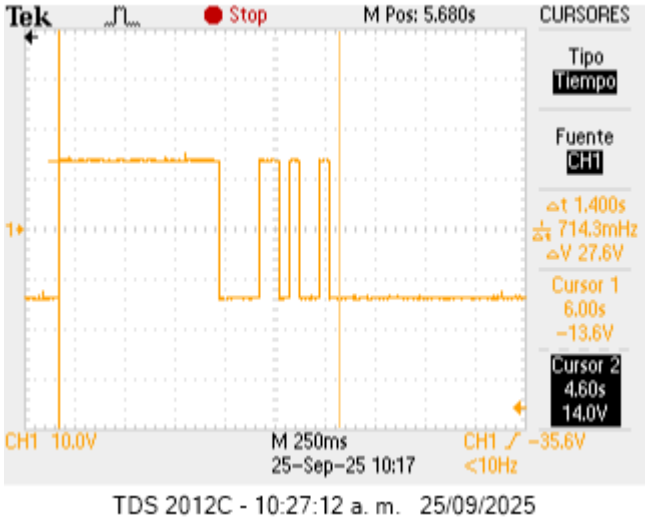

Ilustración 7 captura de la trama en osciloscopio

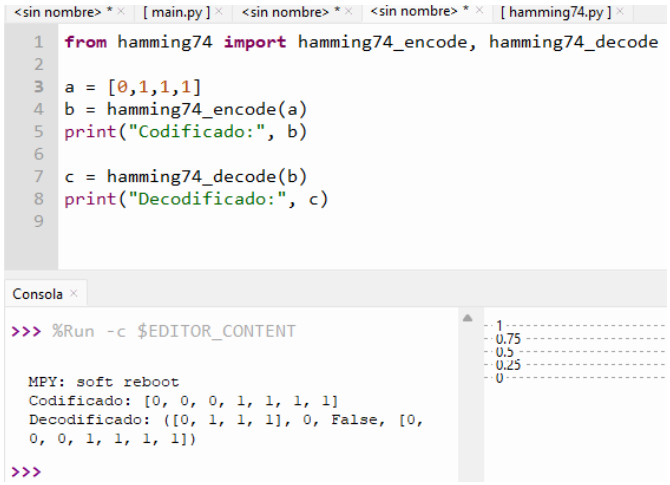

Ilustración 8 codificación y decodificación Hamming (7,4)

En esta prueba se observa la operación correcta del algoritmo Hamming (7,4). El nibble [0,1,1,1] fue convertido en un bloque de 7 bits con los bits de paridad ubicados en sus posiciones correspondientes. Al aplicar la decodificación, el sistema entregó el mismo bloque de datos sin alteraciones, con síndrome igual a cero y sin necesidad de correcciones.

Esto evidencia que el procedimiento de codificación y decodificación implementado es consistente con la teoría: cuando no se presentan errores en la transmisión, los datos se recuperan íntegramente y el decodificador confirma la validez de la trama.

CONCLUSIONES

El laboratorio permitió implementar de manera práctica el código Hamming (7,4) en la Raspberry Pi Pico 2W con el sensor MPU6050, logrando segmentar muestras de 16 bits, codificarlas en tramas de 28 bits y transmitirlas vía UART para su verificación en consola y osciloscopio. Las pruebas realizadas evidenciaron que el sistema mantiene la integridad de los datos y, en presencia de errores de un bit, es capaz de detectarlos y corregirlos automáticamente, lo cual refuerza la importancia de los códigos de bloque en las comunicaciones digitales. De esta manera, se cumplió con los objetivos planteados en la guía, integrando teoría y práctica.

Se implementó con éxito el código Hamming (7,4) en la Raspberry Pi Pico 2W, aplicando la segmentación de muestras y la generación de tramas redundantes.

Las pruebas de codificación y decodificación confirmaron que el sistema es capaz de mantener la integridad de los datos y, en caso de errores, corregirlos en un solo bit.

La transmisión por UART y la visualización en osciloscopio permitieron vincular el código teórico con la señal física, reforzando el aprendizaje práctico en comunicaciones digitales.

Se crea un repositorio en GitHub donde se suben las demás evidencias del desarrollo del laboratorio al cual se podrá acceder en este enlace:

https://github.com/fernandoriano/LABORATORIO3C2--COMUNICACION-DIGITAL.git

III. REFERENCIAS

[1] Universidad Militar Nueva Granada, Guía de laboratorio Corte 2, Lab 3: Codificación Hamming (7,4) con Raspberry Pi Pico 2W, 2025.

[2] MicroPython Development Team. (2024). MicroPython Libraries Overview. [En línea]. Disponible en: https://docs.micropython.org/en/v1.24.0/library/index.html

[3] Raspberry Pi Ltd. (2025). Hardware Design with RP2040 and Pico W. [En línea]. Disponible en: https://www.raspberrypi.com/documentation/microcontrollers/

[4] InvenSense Inc. (2013). MPU-6000 and MPU-6050 Product Specification. [En línea]. Disponible en: https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6050/

[5] R. W. Hamming, “Error Detecting and Error Correcting Codes,” Bell System Technical Journal, vol. 29, no. 2, pp. 147–160, 1950.