ASPECTS OF HAMMING ENCRYPTION USING RASPBERRY PI PICO

Gabriel ANASTASIU[*], **Andreea Valentina ANASTASIU**[**]

[*]”Nationall Coledge Dragos Voda, Campulung Moldovenesc, Romania (agabriel.usv@gmail.com)
[**]Politehnica University, Cluj Napoca, Romania (aanastasiu52@gmail.com)

DOI: 10.19062/2247-3173.2025.26.7

Abstract: *In this article I will try to address the encrypted form of transmission using Hamming encoding on a Raspberyy Pi Pico W platform. I chose to test on a PiPico platform because it offers computing power comparable to that of a microcomputer and allows easy programming in Python.*

Keywords: *Pi Pico, Hamming, bits*

1. INTRODUCTION

Hamming encryption is an error detection and correction technique that uses hamming codes to detect and correct data transmission errors. The Raspberry PI Pico, a microcontroller based on the ARM CORTEX-M0+, is ideal for implementing this type of encryption, having the necessary resources to process and manipulate the data. This microcontroller allows the implementation of subsystems that can make it possible to extend an encryption system for several types of encoding, Fig. 1

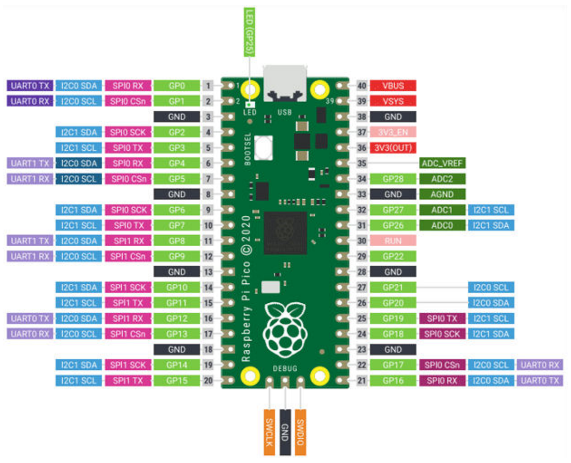


FIG.1 Raspberry Pi Pico Structure

Hamming code is an error-correcting code that adds parity bits to a sequence of data to enable errors to be identified and corrected during data transmission. A hamming code of length 7 (hamming (7,4)) adds 3 bits of parity to a 4-bit message to create a 7-bit sequence, allowing a single bit of error to be detected and corrected.

2. IMPLEMENTING HAMMING ENCRYPTION ON THE RASPBERRY PI PICO

Hamming code uses parity bits to protect data and make it resistant to errors. The 7-bit Hamming code for a 4-bit message has 3 parity bits that are placed on the positions that are powers of 2 (1, 2, 4). The data positions are placed in the other locations.

Example for Hamming code (7,4):

1. Positions 1, 2, and 4 are parity bits.
2. Positions 3, 5, 6, and 7 are data bits.

Encryption process:

- The parity bits are calculated so that the total parity of each set of bits is 0 (even parity).
- After the parity bits are added, the final 7-bit sequence is transmitted.

In Fig. 2 I will present a Hamming encryption system with LCD to be able to visualize the result

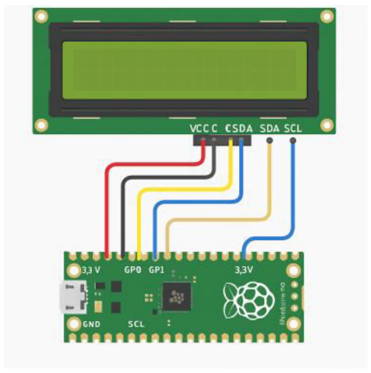
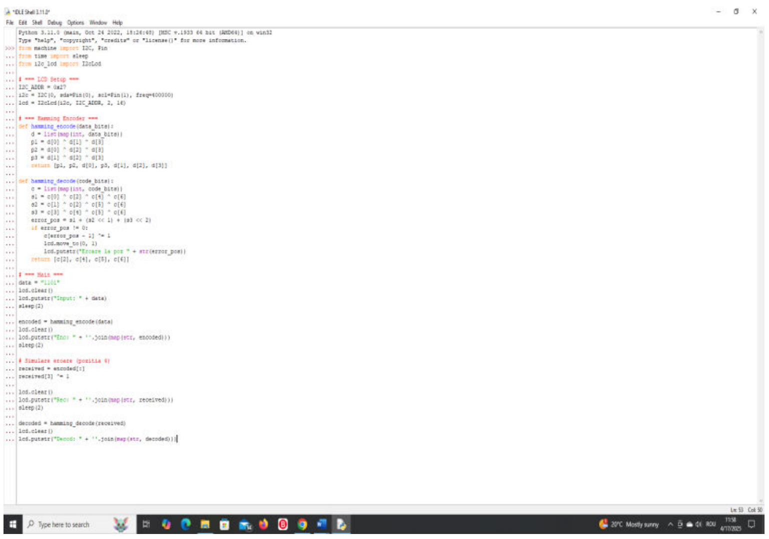


FIG. 2 Schema PICO LCD

I did the programming in Python and is based on the following example, Fig.3:

- Post: 1011
- Parity bits: p1, p2, p4

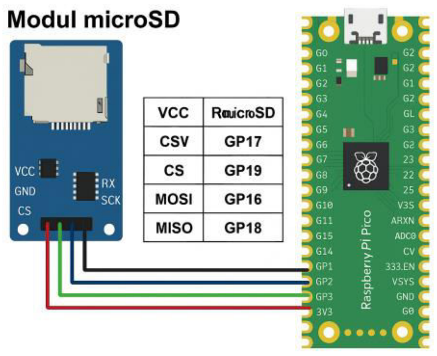
The values for the parity bits are determined so that each parity bit guarantees an even sum of the bits.



```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> from machine import I2C, Pin
... from time import sleep
... from i2c_lcd import I2cLcd
...
... # === LCD Setup ===
... I2C_ADDR = 0x27
... i2c = I2C(0, sda=Pin(0), scl=Pin(1), freq=400000)
... lcd = I2cLcd(i2c, I2C_ADDR, 2, 16)
...
... # === Hamming Encoder ===
... def hamming_encode(data_bits):
...     d = list(map(int, data_bits))
...     p1 = d[0] ^ d[1] ^ d[3]
...     p2 = d[0] ^ d[2] ^ d[3]
...     p3 = d[1] ^ d[2] ^ d[3]
...     return [p1, p2, d[0], p3, d[1], d[2], d[3]]
...
... def hamming_decode(code_bits):
...     c = list(map(int, code_bits))
...     s1 = c[0] ^ c[2] ^ c[4] ^ c[6]
...     s2 = c[1] ^ c[2] ^ c[5] ^ c[6]
...     s3 = c[3] ^ c[4] ^ c[5] ^ c[6]
...     error_pos = s1 + (s2 << 1) + (s3 << 2)
...     if error_pos != 0:
...         c[error_pos - 1] ^= 1
...         lcd.move_to(0, 1)
...         lcd.putstr("Eroare la poz " + str(error_pos))
...     return [c[2], c[4], c[5], c[6]]
...
... # === Main ===
... data = "1101"
... lcd.clear()
... lcd.putstr("Input: " + data)
... sleep(2)
...
... encoded = hamming_encode(data)
... lcd.clear()
... lcd.putstr("Enc: " + ''.join(map(str, encoded)))
... sleep(2)
...
... # Simulare eroare (pozitia 4)
... received = encoded[:]
... received[3] ^= 1
...
... lcd.clear()
... lcd.putstr("Rec: " + ''.join(map(str, received)))
... sleep(2)
...
... decoded = hamming_decode(received)
... lcd.clear()
... lcd.putstr("Decod: " + ''.join(map(str, decoded)))
```

FIG.3 Example of Hamming encoding (7,4)

In order to be able to use the encoding results (e.g. in a radio data transmission system) we can use an SD card save, in which case the data can be kept for a longer period. A model is given in Fig.4 and the code in Fig.3.



VCC	RaicroSD
CSV	GP17
CS	GP19
MOSI	GP16
MISO	GP18

FIG.4 Conectare SD Card la Pi Pico

The connection of the Pi Pico through a radio transmission system can be done using a transmitter and a receiver that can transmit the data in encrypted format. Experimentally we used two HC-12 radio systems, the connection reference is in Fig. 5 and the module is in Fig.6.

HC-12 pin	Pico tx	Pico rx
VCC	3.3v	3.3v
Gnd	Gnd	Gnd
Tx	GP1	GP1
Rx	gp0	gp0

FIG.5 Connection reference

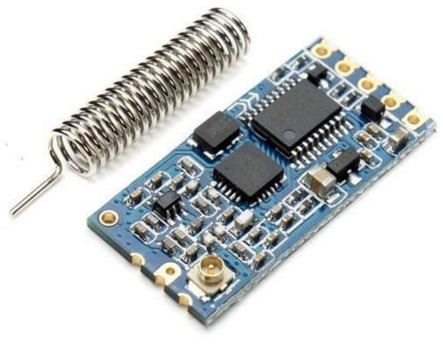
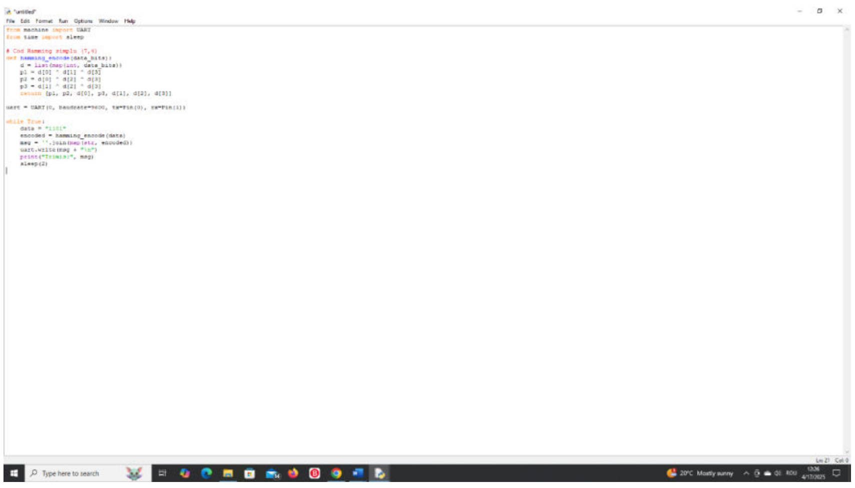


FIG. 6 System Radio HC-12

The corresponding codes for the respective receptive transmission are in Figures 7 and 8 respectively. Same Hammming code was used(7,4)

Hamming_encode function: receives 4 bits of data and calculates the parity bits to form a 7-bit Hamming message.

Hamming_decode function: receives a 7-bit message and checks for errors. If there are errors, it corrects them and returns the original data.



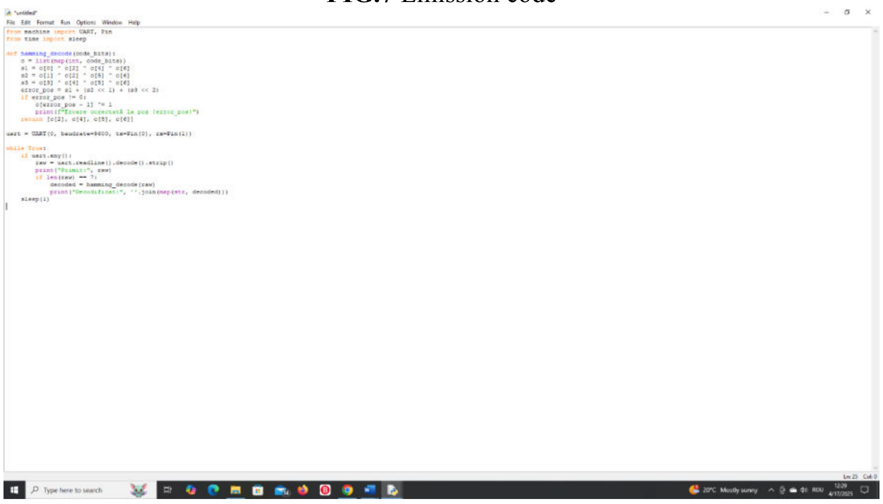
```
from machine import UART
from time import sleep

# Cod Hamming simplu (7,4)
def hamming_encode(data_bits):
    d = list(map(int, data_bits))
    p1 = d[0] ^ d[1] ^ d[3]
    p2 = d[0] ^ d[2] ^ d[3]
    p3 = d[1] ^ d[2] ^ d[3]
    return [p1, p2, d[0], p3, d[1], d[2], d[3]]

uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))

while True:
    data = "1101"
    encoded = hamming_encode(data)
    msg = ''.join(map(str, encoded))
    uart.write(msg + "\n")
    print("Trimis:", msg)
    sleep(2)
```

FIG.7 Emission code



```
from machine import UART, Pin
from time import sleep

def hamming_decode(code_bits):
    c = list(map(int, code_bits))
    s1 = c[0] ^ c[2] ^ c[4] ^ c[6]
    s2 = c[1] ^ c[2] ^ c[5] ^ c[6]
    s3 = c[3] ^ c[4] ^ c[5] ^ c[6]
    error_pos = s1 + (s2 << 1) + (s3 << 2)
    if error_pos != 0:
        c[error_pos - 1] ^= 1
        print(f"Eroare corectată la poz {error_pos}")
    return [c[2], c[4], c[5], c[6]]

uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))

while True:
    if uart.any():
        raw = uart.readline().decode().strip()
        print("Primit:", raw)
        if len(raw) == 7:
            decoded = hamming_decode(raw)
            print("Decodificat:", ''.join(map(str, decoded)))
    sleep(1)
```

FIG.8 Reception Code

3. CONCLUSION

The use of Hamming coding in radio transmission systems is very efficient if such microcontrollers are used, which allow a wide range of applications in Python and can bring an advantage in the case of electronic warfare applications or in amateur radio applications within the emergency networks of Romania. Hamming encryption on the Raspberry Pi Pico is an effective way to protect data and learn about error correction. The implementation can be extended to support more complex applications, such as wireless transmissions or secure data storage.

REFERENCES

[1] S.Monk, *Programming the Pico-Learn Coding and Electronics with the Raspberry Pi Pico*,Technical Editors Ian Huntley and Mike Bassets;

[2] A. Szilagyi, D.S. Stoica, Al. Barbu, D. Cazanaru, *Signal Processing in Non-Communication Electronic Warfare Receivers,* Ferdinand I Military Technical Academy;

[3] D. Coltuc, TTI - *Theory of Information Transmission*, Course Notes 2020.