Project Enigma

Documentation

Summer Project

Electronics Club

Indian Institute of Technology Kanpur

Submitted on June 13, 2025

Mentors:

- Kshitij Bhardwaj
- Tanvi Manhas
- Kamal Jaiswal
- Shrasti Dwivedi

Mentees:

- Abhinav Mishra
- Arushi Kondaskar
- Daksh Gupta
- Dhairya Gupta
- Dhruv Garg
- Dibyanshu Das
- Dipsikha Rano
- Harsh
- Manant Singhal
- Parth Dhamija
- Prakhar Gupta
- Ramya Rasika
- Sahil Agarwal
- Subhankar Mondal
- Vandit Gupta

Abstract

This project is a modern software-based simulation of the historic Enigma Machine, designed to replicate its core encryption mechanism through an interactive and user-friendly interface.

Aim

To recreate the functionality of the Enigma Machine in a digital environment, enabling users to understand and interact with its encryption mechanism through configurable rotor and plugboard settings.

Goals

- **Simulate Historical Encryption Logic:** Implement accurate rotor stepping, letter mapping, and reflection mechanisms.
- **Develop an Interactive Interface:** Provide a GUI for easy configuration of rotors, offsets, and plugboard settings.
- **Enable Real-Time Encryption:** Encrypt user input dynamically, reflecting rotor movements and plugboard mappings.
- **Promote Educational Insight:** Offer a practical tool to help users understand the structure and working of classical cryptographic systems.
- **Ensure User Configurability:** Allow flexibility in rotor selection and position to mimic real-world use of the Enigma Machine.

Contents

1 Introduction to Cryptography

Cryptography is a technique for securing information and communications using codes to ensure confidentiality, integrity, and authentication. Thus, preventing unauthorized access to information. The prefix "crypt" means "hidden" and the suffix "graphy" means "writing". In the past, multiple methods have been used for encrypting methods. Some of these are

1.1 Caesar Cipher

The Caesar Cipher is one of the simplest and oldest encryption techniques. It works by shifting each letter in plain text by a fixed number of positions down the alphabet.

Example : Let us encrypt the word "HELLO" using the shift value to be 3.
H becomes K (shift 3 from H)
E becomes H (shift 3 from E)
L becomes O (shift 3 from L)
L becomes O (shift 3 from L)
O becomes R (shift 3 from O)
Encrypted message : KHOOR

Drawbacks

- **Easily breakable:** Only 25 possible shifts make it vulnerable to brute-force attacks.
- **Frequency patterns remain:** Common letters (like 'E') still appear frequently, enabling frequency analysis.
- **Weak key space:** The key (shift value) is small and easy to guess.
- **Limited scope:** Handles only alphabets unless manually extended.

1.2 Affine Cipher

The **Affine Cipher** is a substitution cipher that uses a mathematical function to encrypt each letter:

$$E(x) = (a \cdot x + b)26$$

Here, x is the numerical value of the letter (A=0, B=1, ...), and a, b are keys such that a and 26 are coprime. Decryption uses the modular inverse of a:

$$D(x) = a^{-1}(x - b)26$$

Example: For $a = 5$, $b = 8$: `HELLO` becomes `RCLLA`.

Drawbacks

- **Limited key space:** Only values of a coprime with 26 are valid, reducing secure options.
- **Susceptible to frequency analysis:** Like Caesar, it preserves letter frequency patterns.
- **Mathematical structure can be exploited:** Known-plaintext attacks can reveal the keys a and b.

1.3 Enigma

The **Enigma Machine** was a highly advanced electro-mechanical cipher device used during World War II. It significantly strengthened encryption by performing multiple layers of letter substitutions that changed with each key press.

Logic Behind Enigma (Step-by-Step with Examples)

1. **Plugboard (Steckerbrett):** Letters are initially swapped using plugboard cables. For example, if A and G are connected:

   $$\texttt{A} \rightarrow \texttt{G}$$

   So, pressing A sends G into the rotor system.

2. **Rotors:** Each rotor maps input letters to outputs via internal wiring. For example, Rotor I might map:

   $$\texttt{G} \rightarrow \texttt{T}$$

   The output depends on the rotor's current position, which shifts over time.

3. **Rotor Stepping:** After each keypress, the rightmost rotor advances by one step. *Example:* If G becomes T now, the next time G may become X due to the rotor's new alignment.

4. **Reflector:** The signal then hits the reflector, which reverses the path. *Example:* If T enters the reflector and is mapped to M:

   $$\texttt{T} \rightarrow \texttt{M}$$

5. **Reverse Path Through Rotors:** The signal travels back through the rotors in reverse, using the inverse mappings. *Example:* M might be mapped back to Y.

6. **Final Plugboard Swap:** The signal goes through the plugboard again. If Y is connected to D:

   $$\texttt{Y} \rightarrow \texttt{D}$$

   So the final encrypted output is `D`.

Thus, even if you press `A` multiple times, the output will vary due to rotor stepping.

Example Summary: Input: `A` After full path: `D` (Next press of `A` could yield `W`, then `Z`, etc.)

Advantages Over Caesar and Affine

- **Dynamic Substitution:** The rotor mechanism changes the encryption path with every keystroke.
- **Huge Key Space:** Over 10^{114} possible configurations with rotors, positions, and plugboard.
- **Highly Resistant to Frequency Analysis:** Because the same letter can encrypt differently every time.

Drawbacks

- **Operational Complexity:** Required correct daily rotor and plugboard settings to function properly.
- **Key Reuse Weakness:** Predictable patterns and operator errors led to its eventual cryptanalysis.

2 Objectives

2.1 Software-Based Enigma Simulator

The objectives of the software-based Enigma simulator project are as follows:

1. **Design and implement a fully functional Enigma machine simulator** that replicates the historical operation of the original device, including rotor stepping, plugboard substitution, and reflection.
2. **Implement all core components in code:**
   - Plugboard with customizable letter pairings
   - Multiple rotors with configurable wiring and turnover positions
   - Reflector that ensures reversibility of the cipher
3. **Enable user customization of settings:** Allow the selection of rotor types, rotor order, initial rotor positions, and plugboard configurations.
4. **Develop a graphical user interface (GUI)** using JavaScript and PySimple GUI to simulate Enigma's keyboard and lampboard.
5. **Ensure symmetric encryption/decryption:** Verify that messages encrypted with a particular configuration can be correctly decrypted with the same configuration.
6. **Provide import/export functionality:** Allow users to save and load machine settings and encrypted messages.
7. **Focus on modular and maintainable code:** Use object-oriented principles and modular programming to facilitate future extensions or improvements.

2.2 Hardware-Based Enigma Replica

The objectives of the hardware-based Enigma project are:

1. **Construct a physical replica of the Enigma machine** using microcontrollers such as Arduino or PIC32, closely emulating the functionality of the original.
2. **Implement mechanical components:**
   - Use push buttons or a physical keyboard for input
   - Simulate rotors using motors or LED indicators
   - Display output using LEDs to represent the lampboard
3. **Program rotor logic and stepping mechanism:** Replicate rotor movement, turnover, and encryption logic using embedded programming .
4. **Develop a functional plugboard interface:** Simulate the plugboard using jumper wires, switches, or a digital interface for changing letter pairings.

5. **Ensure operational accuracy:** Validate that the encryption and decryption processes function identically to the original machine.

6. **Design an enclosure or case:** Optionally, construct or 3D print a physical housing to resemble the layout of the historical Enigma machine.

3 Online Simulation

At the very beginning of the project we started with an assignment taken from Stanford University. The assignment was to complete the Enigma simulation in JavaScript.

Link : https://web.stanford.edu/class/cs106j/handouts/37-Assignment5.pdf

3.1 Java Script Code

EnigmaConstants.js – This file defines the constants used in the Enigma simulator

This section documents the UI layout and core logic of the rotor-based Enigma simulator. It includes configuration parameters for rotors, keyboard, lamps, and signal flow behavior, designed to resemble a simplified Enigma machine.

Rotor Configuration

Each rotor uses a unique permutation of the alphabet to scramble input signals. Three such permutations represent the slow, medium, and fast rotors:

```
const ROTOR_PERMUTATIONS = [
  "EKMFLGDQVZNTOWYHXUSPAIBRCJ",  // Slow rotor
  "AJDKSIRUXBLHWTMCQGZNPYFVOE",  // Medium rotor
  "BDFHJLCPRTXVZNYEIWGAKMUSQO"   // Fast rotor
];
```

UI Parameters: The rotors are styled and placed using constants for color, size, and position.

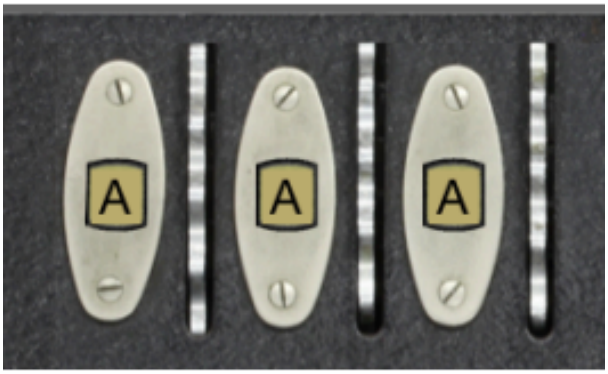

Figure 1: Rotor placement and visual offset

```
const ROTOR_LOCATIONS = [
   { x: 244, y: 95 },
   { x: 329, y: 95 },
   { x: 412, y: 95 }
];
```

Other constants like ‘ROTOR BGCOLOR‘, ‘ROTOR WIDTH‘, and ‘ROTOR FONT‘ define the rotor’s appearance.

Reflector Configuration

The reflector provides a fixed symmetric mapping of letters. This ensures that encryption and decryption are mirror operations.

```
const REFLECTOR_PERMUTATION = "IXUHFEZDAOMTKQJWNSRLCYPBVG";
```

Keyboard UI (Key Press Input)

The keyboard consists of 26 circular keys laid out visually. Each key changes color based on its state (up or down).

Figure 2: Key layout

Key Parameters:

- `KEY_RADIUS`, `KEY_BGCOLOR`, `KEY_UP_COLOR`, `KEY_DOWN_COLOR`
- `KEY_LOCATIONS[26]`: Predefined `x, y` positions for each key

Behavior: Keys are generated in a loop over `KEY_LOCATIONS`, labeled from A to Z. When clicked, the key triggers the encryption process.

Lampboard UI (Output)

Lamps are positioned similarly to keys. They light up in yellow when the encrypted character is computed.

Figure 3: Lampboard UI

Lamp Parameters:

- `LAMP_RADIUS`, `LAMP_ON_COLOR`, `LAMP_OFF_COLOR`, `LAMP_FONT`
- `LAMP_LOCATIONS[26]`: Matches key layout for consistent mapping

Behavior: Lamps are rendered in a loop and light up when their corresponding character is the encryption output.

Signal Flow Logic (Pseudocode)

The following pseudocode summarizes the flow of a single character through the machine:

```
function encryptInput(inputChar):
    index = charToIndex(inputChar)

    // Rotate rightmost rotor and propagate if needed
    rotateRotors()

    // Forward pass through rotors (R -> L)
    for rotor in reverse(rotors):
        index = rotor.forward(index)

    // Reflector mapping
    index = reflector.reflect(index)

    // Backward pass through rotors (L -> R)
    for rotor in rotors:
        index = rotor.backward(index)

    outputChar = indexToChar(index)
    highlightLamp(outputChar)
```

Note: Rotor stepping logic is triggered before each key press and follows traditional Enigma stepping (cascade).

Suggested Architecture Overview (Optional)

To better understand the system layout, consider visualizing the following pipeline:

- **Input:** Key pressed → Rotor rotation
- **Processing:** Input flows through rotors → reflector → back through rotors
- **Output:** Result shown by lighting up the corresponding lamp

Enigma.js — Graphical Simulation of the Enigma Machine

This section documents the JavaScript simulation of the Enigma machine using Stanford's Graphics.js. It visually represents the machine's rotors, keys, and lamps, providing an interactive encryption experience.

Initialization

The simulation begins by loading an image of the Enigma machine into a 'GWindow'. Then, it calls the main simulation function.

```
function Enigma() {
   var enigmaImage = GImage("EnigmaTopView.png");
   var gw = GWindow(enigmaImage.getWidth(), enigmaImage.getHeight
      ());
   gw.add(enigmaImage);
   runEnigmaSimulation(gw);
}
```

—

Rotor Rotation (Double-stepping logic)
Pseudocode:

```
Increment right rotor.
If middle rotor is at notch, advance middle and left rotor.
If right rotor wraps, also increment middle rotor.
Update rotor display letters.
```

—

Encrypting a Character
Pseudocode:

```
index = A-Z index of letter
For each rotor (right to left):
    Apply rotor offset
    Map letter forward using permutation
Reflect the signal
For each rotor (left to right):
    Map letter backward using inverse permutation
Apply reverse rotor offset
Rotate rotors
Return encrypted character
```

—

Rotor Construction
Each rotor is a rectangle with a label inside a compound object. The label updates to show the current offset letter.
Pseudocode:

```
For i = 0 to 2:
    Create rectangle and label ("A")
    Add to GCompound
    Set setRotor(index) function to update label
    Append to enigma.rotors[]
```

—

Key and Lamp Construction
Each key/lamp is a circular component with a letter label. Actions are attached:
Pseudocode (Key setup):

```
For each letter A-Z:
    Create outer and inner circle
    Add label
    Add to GCompound
    key.mousedownAction: highlight key
    key.mouseupAction: reset key
    Add to enigma.keys[]
```

Pseudocode (Lamp setup):

```
Create lamp circle and label
Define lamp.turnOn() and lamp.turnOff()
Add to enigma.lamps[]
```

—

Mouse Event Listeners

Two event listeners handle user interaction:

on mousedown:

```
If a rotor is clicked:
    Rotate it, update label

If a key is clicked:
    Turn off previous lamp
    Store current key
    Call mousedownAction()
```

on mouseup:

```
If a key was held:
    Get its letter
    Encrypt the letter
    Find lamp for encrypted letter
    Turn it on
    Call key.mouseupAction()
```

—

This structured simulation not only reflects the logic of rotor-based encryption, but also integrates interactive UI behavior, closely mimicking the historical Enigma machine.

4 Electronics Components

4.1 Microcontroller: Arduino Integration

4.1.1 Overview

The **Arduino Nano or Uno** microcontroller serves as the central unit responsible for handling encrypted communication between the software interface and the hardware display system. It interprets input data received from the GUI via serial communication, performs encryption logic, updates the rotor settings, and displays the resulting ciphertext on a 16x2 LCD display.

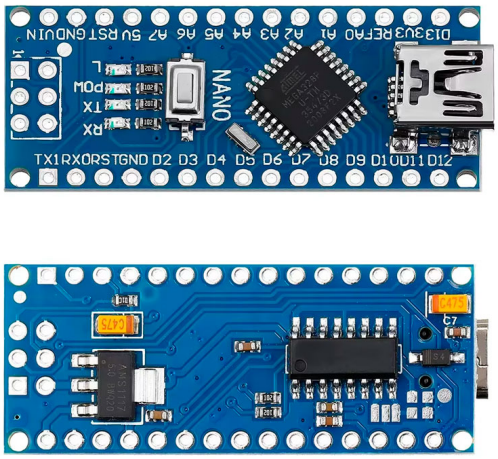

Figure 4: Arduino Nano Microcontroller

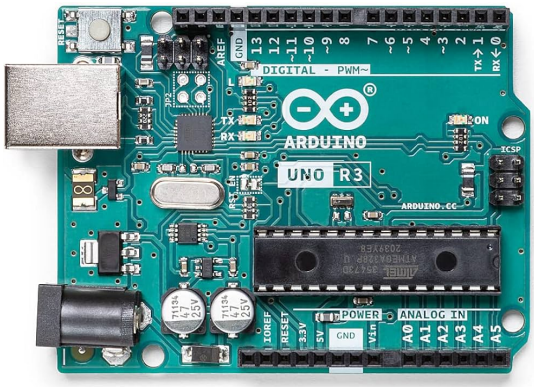

Figure 5: Arduino Uno Microcontroller

4.1.2 Responsibilities of the Arduino

- Receiving plaintext and rotor configuration data from the PC using serial communication.
- Executing the Enigma encryption algorithm.

- Displaying the resulting ciphertext on a 16x2 character LCD.
- Monitoring a rotary encoder to adjust rotor offsets in real-time.

4.2 LCD Display Interface

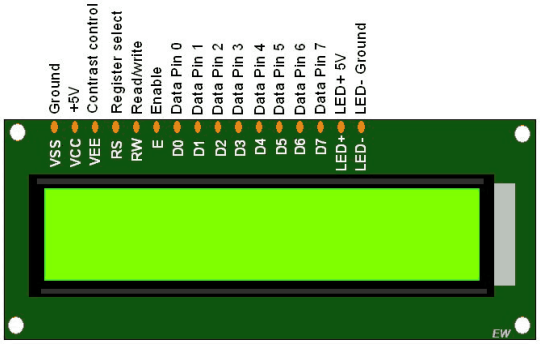

Figure 6: 16x2 Character LCD Display

The Arduino is connected to a 16x2 LCD display (via I2C or 4-bit parallel mode). It uses the `LiquidCrystal` or `LiquidCrystal_I2C` library to:

- Display the current rotor settings.
- Show encrypted characters as they are generated.
- Provide feedback such as rotor selection or system status.

4.3 Rotary Encoder for Rotor Offset Control

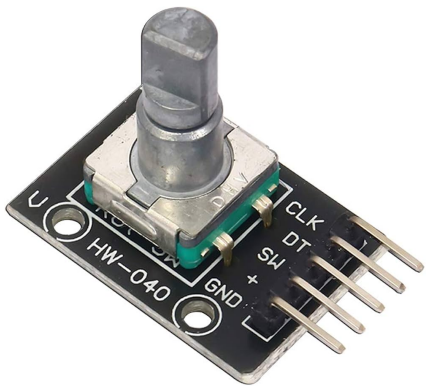

Figure 7: Rotary Encoder with Push Button

The rotary encoder is interfaced with digital pins on the Arduino to allow manual adjustment of rotor positions:

- Rotating the encoder changes the offset of the currently selected rotor.
- Pressing the encoder switch cycles between Rotor I, II, and III.

- The Arduino updates the LCD to reflect the selected rotor and its offset.

The encoder is read using either polling or interrupts, depending on the desired responsiveness and complexity.

4.4 Serial Communication Protocol

The Arduino listens for serial input in a specific format, for example:

```
R1=A;R2=B;R3=C;MSG=HELLO
```

- Rotor positions (R1–R3) and message content are parsed.
- Each character of the message is encrypted using the current configuration.
- The encrypted output is written to the LCD display.

4.5 Conclusion

The Arduino acts as the bridge between the software-driven GUI and the physical interface of the Enigma machine. Its integration enables real-time encryption feedback, dynamic rotor control, and a hands-on educational experience in both cryptography and embedded systems.

4.6 Circuit and PCB Design

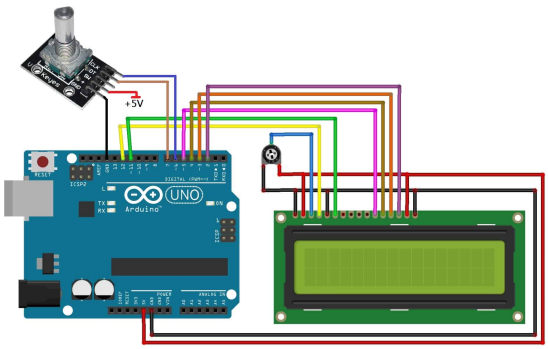

Figure 8: Circuit Diagram of Rotary Encoder, LCD with Arduino Uno

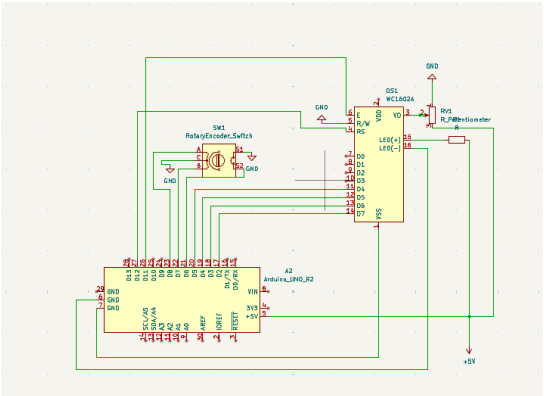

Figure 9: Schematic

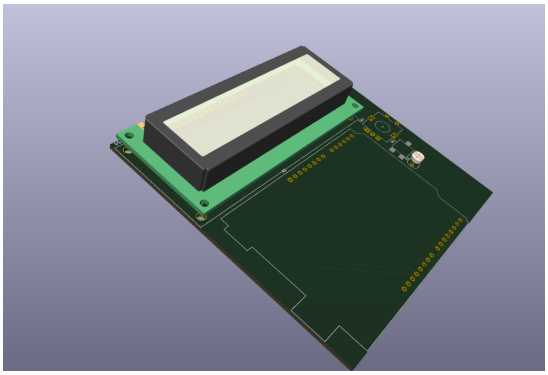

Figure 10: PCB

5 Software

5.1 Protothreads

Definition and Analogy: What are Protothreads?

Definition: Protothreads are extremely lightweight, stackless threads (coroutines) that enable cooperative multitasking in C, especially useful for memory-constrained embedded systems.

Analogy: Think of protothreads as a *traffic officer* managing cars (tasks) at a single-lane intersection (CPU). Each car (task) must wait until the officer (scheduler) signals it to move. There is no chaos, just orderly movement controlled by explicit yielding and resuming.

Real-World Concept	Protothread Equivalent	Explanation
Cars	Protothreads	Each task is like a car waiting to move through the CPU
Traffic Officer	Scheduler	Decides which thread gets to execute next
Green Light	Condition met / Scheduled	Allows a thread to continue
Stop Sign	PT_WAIT_UNTIL / PT_YIELD	The thread pauses here, waiting for the next signal
Intersection	CPU	Only one car (thread) moves at a time

5.1.1

How Protothreads Work

- Implemented via macros like `PT_BEGIN`, `PT_END`, `PT_YIELD`, and `PT_WAIT_UNTIL`.
- Do not require individual stacks; memory usage is minimal.
- Written as linear code, unlike traditional state machines.

Example Code:

```
PT\_THREAD(thread\_example(struct pt \*pt)) {
PT\_BEGIN(pt);
while(1) {
PT\_WAIT\_UNTIL(pt, condition\_met);
execute\_task();
PT\_YIELD(pt);
}
PT\_END(pt);
}
```

5.1.2

System Example: Encryption with Protothreads

An embedded encryption system using protothreads may include:

- **Serial Communication Thread:** Waits for data from UART.
- **Encryption Thread:** Encrypts incoming strings.
- **Display Thread:** Updates a TFT screen.
- **Encoder Thread:** Reads rotary input to set rotor values.

Each of these threads shares data through global buffers and flags, coordinated cooperatively.

5.1.3

Scheduling with Protothreads

Types of Scheduling

- **Round-Robin:** Each thread is called in sequence.
- **Rate-Controlled:** Threads run at different loop intervals.
- **Event-Driven:** Threads resume when conditions are met.

Rate-Controlled Example:

```
\#define MAX\_THREADS 10
struct ptx {
struct pt pt;
int rate;
char (\*pf)(struct pt \*pt);
};

while(1) {
loop\_count++;
for(int i = 0; i < pt\_task\_count; i++) {
if ((loop\_count % (1 << pt\_thread\_list\[i].rate)) == 0) {
(pt\_thread\_list\[i].pf)(\&pt\_thread\_list\[i].pt);
}
}
}
```

5.1.4

Advantages and Limitations

Advantages:

- Extremely memory-efficient (1-2 bytes per thread).
- Enables clear, readable, sequential-style code.
- Ideal for simple scheduling in embedded systems.

Limitations:

- Cannot retain non-static local variables across yields.
- No built-in preemption; tasks must yield explicitly.

5.1.5

Summary

Protothreads provide an elegant and efficient way to manage multiple tasks on microcontrollers. Using the traffic officer analogy helps simplify their operation: cooperative, controlled multitasking where every task waits its turn and resumes only when allowed. This model supports encryption systems, sensor networks, and other embedded tasks without the overhead of full threading models.

5.2 Enigma Logic Code

5.2.1 Code in C

The code begins with standard header files and macro definitions for the number of rotors and the alphabet size. These are essential for configuring the simulation and for indexing operations.

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>

#define NUM_ROTORS 3
#define ALPHABET_SIZE 26
```

The rotor wirings are stored as fixed strings for Rotor I, II, and III. The reflector wiring corresponds to Reflector B used in the historical Enigma machine.

```c
char rotors[NUM_ROTORS][ALPHABET_SIZE + 1] = {
    "EKMFLGDQVZNTOWYHXUSPAIBRCJ",
    "AJDKSIRUXBLHWTMCQGZNPYFVOE",
    "BDFHJLCPRTXVZNYEIWGAKMUSQO"
};

char reflector[ALPHABET_SIZE + 1] = "YRUHQSLDPXNGOKMIEBFZCWVJAT";
```

The rotor offsets array keeps track of the current rotation of each rotor. turnovers define the position at which each rotor causes the one to its left to step pairings and sketcherboard define letter swaps for plugboard and the experimental sketcherboard.

```c
volatile int rotor_offsets[NUM_ROTORS] = {0, 0, 0};
int turnovers[NUM_ROTORS] = {'Q' - 'A', 'E' - 'A', 'V' - 'A'};

int pairings[ALPHABET_SIZE] = {0};
int sketcherboard[ALPHABET_SIZE] = {0};
bool sketcher_board_on = false;

int stepping[NUM_ROTORS] = {0, 0, 0};

char input_buffer[256] = {0};
char output_buffer[256] = {0};
```

These functions convert between letters and their corresponding 0–25 index values, which are used throughout the logic.

```c
int char_to_index(char c) {
    return toupper(c) - 'A';
}

char index_to_char(int index) {
    return 'A' + (index % ALPHABET_SIZE);
}
```

Signals pass through the rotors in two directions. These functions simulate the rotor's forward (‘rotor$_{r}to_{l}$‘)$and backward($‘$rotor_{l}to_{r}$‘)$traversal, considering the rotor's current offset.$

```c
int index_inverse(int c, int rotor) {
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (rotors[rotor][i] == c + 'A') {
            return i;
```

```
        }
    }
    return -1;
}

int rotor_r_to_l(int input, int rotor) {
    int idx = (input + rotor_offsets[rotor]) % ALPHABET_SIZE;
    int mapped = rotors[rotor][idx] - 'A';
    int res = mapped - rotor_offsets[rotor];
    if (res < 0) res += ALPHABET_SIZE;
    if (res >= ALPHABET_SIZE) res -= ALPHABET_SIZE;
    return res;
}

int rotor_l_to_r(int input, int rotor) {
    int idx = (input + rotor_offsets[rotor]) % ALPHABET_SIZE;
    int inverse = index_inverse(idx, rotor);
    int res = inverse - rotor_offsets[rotor];
    if (res < 0) res += ALPHABET_SIZE;
    if (res >= ALPHABET_SIZE) res -= ALPHABET_SIZE;
    return res;
}
```

The reflector bounces the signal back through the rotors after the forward traversal, ensuring symmetric encryption.

```
int reflect(int input) {
    return reflector[input] - 'A';
}
```

These functions parse user-input plugboard and sketcherboard pairs and initialize the substitution mappings accordingly.

```
void initialize_plugboard(const char *pairs) {
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        pairings[i] = 0;
    }
    for (int i = 0; i < strlen(pairs); i += 3) {
        if (pairs[i+1] == ' ' && i+2 < strlen(pairs)) {
            int first = toupper(pairs[i]) - 'A';
            int second = toupper(pairs[i+2]) - 'A';
            if (first >= 0 && first < ALPHABET_SIZE && second >=
               0 && second < ALPHABET_SIZE) {
                pairings[first] = second - first;
                pairings[second] = first - second;
            }
        }
    }
}

void initialize_sketcherboard(const char *pairs) {
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        sketcherboard[i] = 0;
```

```
    }
    for (int i = 0; i < strlen(pairs); i += 3) {
        if (pairs[i+1] == ' ' && i+2 < strlen(pairs)) {
            int first = toupper(pairs[i]) - 'A';
            int second = toupper(pairs[i+2]) - 'A';
            if (first >= 0 && first < ALPHABET_SIZE && second >=
               0 && second < ALPHABET_SIZE) {
                sketcherboard[first] = second - first;
                sketcherboard[second] = first - second;
            }
        }
    }
}
```

5.2.2 Code in Python

This Python script reuses the same Enigma machine logic as the C version, allowing encryption through rotor and plugboard-based transformation. However, it adds one key feature: the encrypted message is sent over a USB serial connection to an Arduino, which can display it on an OLED or LCD screen. This extension shifts the project from a pure software simulation to hardware-based real-world interaction.

```
import string
import serial
import time

ALPHABET = string.ascii_uppercase
ALPHABET_SIZE = 26

rotors = [
    "EKMFLGDQVZNTOWYHXUSPAIBRCJ",  # Rotor I
    "AJDKSIRUXBLHWTMCQGZNPYFVOE",  # Rotor II
    "BDFHJLCPRTXVZNYEIWGAKMUSQO"   # Rotor III
]

reflector = "YRUHQSLDPXNGOKMIEBFZCWVJAT"
turnovers = [ord('Q') - 65, ord('E') - 65, ord('V') - 65]

rotor_offsets = [0, 0, 0]
plugboard_map = {ch: ch for ch in ALPHABET}
```

Same Enigma wiring and setup as the C implementation.

```
def char_to_index(c):
    return ord(c) - ord('A')

def index_to_char(i):
    return ALPHABET[i % ALPHABET_SIZE]

def set_rotor_positions(left, middle, right):
    rotor_offsets[2] = char_to_index(left)
    rotor_offsets[1] = char_to_index(middle)
```

```
    rotor_offsets[0] = char_to_index(right)

def initialize_plugboard(pairs):
    global plugboard_map
    plugboard_map = {ch: ch for ch in ALPHABET}
    tokens = pairs.upper().split()
    for i in range(0, len(tokens), 2):
        if i + 1 < len(tokens):
            a, b = tokens[i], tokens[i + 1]
            plugboard_map[a], plugboard_map[b] = b, a

def plug_swap(c):
    return plugboard_map.get(c, c)
```

Basic character mapping and user input setup logic.

```
def spin_rotors():
    r, m, l = rotor_offsets
    will_middle_step = (m == turnovers[1])
    will_right_step = (r == turnovers[0])

    rotor_offsets[0] = (r + 1) % ALPHABET_SIZE

    if will_right_step or will_middle_step:
        rotor_offsets[1] = (m + 1) % ALPHABET_SIZE

    if will_middle_step:
        rotor_offsets[2] = (l + 1) % ALPHABET_SIZE

def rotor_forward(c, rotor_index, offset):
    idx = (char_to_index(c) + offset) % ALPHABET_SIZE
    wired = rotors[rotor_index][idx]
    return index_to_char((char_to_index(wired) - offset) %
       ALPHABET_SIZE)

def rotor_backward(c, rotor_index, offset):
    idx = (char_to_index(c) + offset) % ALPHABET_SIZE
    char = ALPHABET[idx]
    pos = rotors[rotor_index].index(char)
    return index_to_char((pos - offset) % ALPHABET_SIZE)

def reflect(c):
    return reflector[char_to_index(c)]
```

Rotor traversal and reflector logic — identical to C logic.

```
def encrypt_char(c):
    if not c.isalpha():
        return c
    c = c.upper()

    spin_rotors()

```

```
    c = plug_swap(c)
    c = rotor_forward(c, 0, rotor_offsets[0])
    c = rotor_forward(c, 1, rotor_offsets[1])
    c = rotor_forward(c, 2, rotor_offsets[2])
    c = reflect(c)
    c = rotor_backward(c, 2, rotor_offsets[2])
    c = rotor_backward(c, 1, rotor_offsets[1])
    c = rotor_backward(c, 0, rotor_offsets[0])
    c = plug_swap(c)

    return c

def encrypt_message(message):
    result = ""
    for ch in message:
        if ch == ' ':
            result += ' '
            while True:
                choice = input("Space encountered. Update
                   plugboard? (y/n): ").strip().lower()
                if choice == 'y':
                    pairs = input("Enter new plugboard pairs (e.g
                       ., A B C D): ")
                    initialize_plugboard(pairs)
                    print("Plugboard updated.")
                    break
                elif choice == 'n':
                    break
        else:
            result += encrypt_char(ch)
    return result
```

Full message encryption with optional runtime plugboard update.

```
def print_rotor_status():
    print(f"Rotor positions (L M R): {index_to_char(rotor_offsets
       [2])} "
          f"{index_to_char(rotor_offsets[1])} {index_to_char(
             rotor_offsets[0])}")
```

Displays current rotor positions.

```
def send_to_arduino(message):
    try:
        arduino = serial.Serial('/dev/tty.usbmodem1301', 9600,
           timeout=1)
        time.sleep(2)  # Wait for Arduino to initialize
        arduino.write((message + '\n').encode('utf-8'))
        arduino.close()
        print("Encrypted message sent to Arduino OLED.")
    except Exception as e:
        print(f"Error sending to Arduino: {e}")
```

This is the key addition in the Python version. After encryption, the full message is sent over serial (USB) to an Arduino. On the Arduino, the received message can be displayed on a screen, such as an OLED. Make sure to match the port name (like ‘’/dev/tty.usbmodem1301‘‘) to your system.

```
115 def main():
116     print("=== Enigma Machine Simulator ===")
117     set_rotor_positions('A', 'A', 'A')
118 
119     while True:
120         print_rotor_status()
121         print("\n1. Set rotor positions")
122         print("2. Set plugboard pairs")
123         print("3. Encrypt message")
124         print("4. Exit\n")
125 
126         cmd = input("Enter option: ").strip()
127         if cmd == '1':
128             pos = input("Enter positions (L M R): ").upper().
                split()
129             if len(pos) == 3 and all(p in ALPHABET for p in pos):
130                 set_rotor_positions(pos[0], pos[1], pos[2])
131                 print("Rotors updated.")
132             else:
133                 print("Invalid input.")
134         elif cmd == '2':
135             pairs = input("Enter plugboard pairs: ")
136             initialize_plugboard(pairs)
137         elif cmd == '3':
138             message = input("Enter message to encrypt: ")
139             encrypted = encrypt_message(message)
140             print(f"Encrypted: {encrypted}")
141             send_to_arduino(encrypted)
142         elif cmd == '4':
143             break
144         else:
145             print("Invalid option.")
```

The main menu loop where the user selects an action and encrypted output gets sent to Arduino.

```
146 if __name__ == "__main__":
147     main()
```

Runs the program only if executed directly.

5.3 Enigma GUI Code

This Python script builds a graphical user interface (GUI) for the Enigma simulator using PySimpleGUI. It allows the user to input a message, choose rotor positions, configure plugboard pairs, and encrypt messages with the click of a button. Internally, it reuses the Enigma logic already defined in the earlier C and Python scripts. Once the encryption

is done, the encrypted message is also sent via serial to an Arduino board for external display on an OLED or similar device.

```
import serial
import time
import PySimpleGUI as sg

print(sg.__file__)
print(sg.__version__)
```

Imports required modules: ‘serial‘ and ‘time‘ for Arduino interaction, and ‘PySimpleGUI‘ for the user interface.

```
rotors = [
    "EKMFLGDQVZNTOWYHXUSPAIBRJC",
    "AJDKSIRUXBLHWTMCQGZNPYFVOE",
    "BDFHJLCPRTXVZNYEIWGAKMUSQO"
]
reflector = "IXUHFEZDAOMTKQJWNSRLCYPBVG"
turnovers = [ord('Q') - 65, ord('E') - 65, ord('V') - 65]
rotor_offsets = [0, 0, 0]

plugboard = [i for i in range(26)]
```

Defines rotor wirings, reflector, turnover notches, and default rotor offsets. The plugboard is initialized with identity mapping.

```
def char_to_index(c):
    return ord(c.upper()) - 65

def index_to_char(i):
    return chr((i % 26) + 65)
```

Helper functions to convert characters to/from 0–25 indexing.

```
def rotor_r_to_l(c, r):
    index = (c + rotor_offsets[r]) % 26
    mapped = ord(rotors[r][index]) - 65
    return (mapped - rotor_offsets[r]) % 26

def rotor_l_to_r(c, r):
    index = (c + rotor_offsets[r]) % 26
    inverse = rotors[r].index(index_to_char(index))
    return (inverse - rotor_offsets[r]) % 26

def reflect(c):
    return ord(reflector[c]) - 65
```

Same rotor traversal and reflector logic as the core engine.

```
def plug_swap(c):
    return plugboard[c]

def initialize_plugboard(pairs):
    global plugboard
```

```
    plugboard = list(range(26))
    if not pairs.strip():
        return
    pairs = pairs.upper().split()
    for i in range(0, len(pairs), 2):
        if i+1 >= len(pairs):
            break
        a, b = char_to_index(pairs[i]), char_to_index(pairs[i+1])
        plugboard[a], plugboard[b] = b, a
```

Configures the plugboard mapping based on user input.

```
def spin_rotors():
    middle_at_notch = rotor_offsets[1] == turnovers[1]
    right_at_notch = rotor_offsets[0] == turnovers[0]
    if middle_at_notch:
        rotor_offsets[2] = (rotor_offsets[2] + 1) % 26
        rotor_offsets[1] = (rotor_offsets[1] + 1) % 26
    elif right_at_notch:
        rotor_offsets[1] = (rotor_offsets[1] + 1) % 26
    rotor_offsets[0] = (rotor_offsets[0] + 1) % 26
```

Implements rotor stepping mechanism, including double-stepping.

```
def encrypt_char(c):
    if not c.isalpha():
        return c
    c = char_to_index(c)
    c = plug_swap(c)
    spin_rotors()
    c = rotor_r_to_l(c, 2)
    c = rotor_r_to_l(c, 1)
    c = rotor_r_to_l(c, 0)
    c = reflect(c)
    c = rotor_l_to_r(c, 0)
    c = rotor_l_to_r(c, 1)
    c = rotor_l_to_r(c, 2)
    c = plug_swap(c)
    return index_to_char(c)
```

Encrypts a single character using the full Enigma path.

```
def encrypt_message(message):
    result = ""
    for ch in message:
        if ch == ' ':
            result += ' '
            while True:
                choice = input("Space encountered. Update
                   plugboard? (y/n): ").strip().lower()
                if choice == 'y':
                    pairs = input("Enter new plugboard pairs (e.g
                       ., A B C D): ")
                    initialize_plugboard(pairs)
```

```
                    break
                elif choice == 'n':
                    break
        else:
            result += encrypt_char(ch)
    return result
```

Encrypts a message and optionally allows plugboard update on space.

```
def set_rotor_positions(left, middle, right):
    rotor_offsets[2] = char_to_index(left)
    rotor_offsets[1] = char_to_index(middle)
    rotor_offsets[0] = char_to_index(right)
```

Sets the initial rotor positions.

```
PLACEHOLDER = 'A-A, B-B, C-C, ...'
alphabet = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
layout = [
    [sg.Text("Enter Message to Encrypt:")],
    [sg.Input(key='-INPUT-', size=(40, 1))],
    [sg.Text("Plugboard configuration:")],
    [sg.Input(key='-PLUG-', size=(40,1), default_text=PLACEHOLDER
       )],
    [sg.Text("Rotor Positions:")],
    [sg.Column([
        [sg.Text('Rotor 1', background_color="#3DA9F0"),
         sg.Text('Rotor 2', background_color='#3DA9F0'),
         sg.Text('Rotor 3', background_color='#3DA9F0')],
        [sg.Listbox(values=alphabet, size=(8, 6), key='R1'),
         sg.Listbox(values=alphabet, size=(8, 6), key='R2'),
         sg.Listbox(values=alphabet, size=(8, 6), key='R3')]
    ])],
    [sg.Button("Encrypt", key='-ENCRYPT-'), sg.Button("Exit")],
    [sg.Output(size=(60, 15), key='-OUTPUT-')]
]
```

Defines the full PySimpleGUI layout — input fields, rotor selectors, buttons, and output box.

```
current_r1 = 'A'
current_r2 = 'A'
current_r3 = 'A'
window = sg.Window("Enigma Simulator", layout)
```

Initializes default rotor settings and opens the GUI window.

```
while True:
    event, values = window.read(timeout=100)
    if event == sg.WINDOW_CLOSED or event == "Exit":
        break

    if event == '-PLUG-':
        if values['-PLUG-'] == PLACEHOLDER:
```

```
            window['-PLUG-'].update(value="", text_color='black')

    if event == '-ENCRYPT-':
        message = values['-INPUT-'].upper()
        plug_pairs = values['-PLUG-'].strip()
        if plug_pairs == PLACEHOLDER:
            plug_pairs = ""
        r1 = values['R1'][0] if values['R1'] else 'A'
        r2 = values['R2'][0] if values['R2'] else 'A'
        r3 = values['R3'][0] if values['R3'] else 'A'

        set_rotor_positions(current_r1, current_r2, current_r3)
        initialize_plugboard(plug_pairs)

        encrypted = ""
        for ch in message:
            if ch == ' ':
                new_pairs = sg.popup_get_text("Update plugboard
                   or press OK:", default_text=plug_pairs)
                if new_pairs:
                    plug_pairs = new_pairs
                initialize_plugboard(plug_pairs)
                encrypted += ' '
            else:
                encrypted += encrypt_char(ch)

        print(f"\nOriginal:  {message}")
        print(f"Encrypted: {encrypted}")

        current_r1 = index_to_char(rotor_offsets[2])
        current_r2 = index_to_char(rotor_offsets[1])
        current_r3 = index_to_char(rotor_offsets[0])

        window['R1'].update(set_to_index=rotor_offsets[0])
        window['R2'].update(set_to_index=rotor_offsets[1])
        window['R3'].update(set_to_index=rotor_offsets[2])
```

Main GUI loop — reads inputs, runs encryption, updates rotor displays, and prints result.

```
window.close()
```

Closes the GUI window when the user exits.

```
arduino = serial.Serial(port="COM3", baudrate=9600, timeout=1)
time.sleep(2)
message = "One piece is real"
arduino.write((message + '\n').encode())
arduino.close()
```

At the end, sends a sample message over serial to Arduino. This part can be modified to send the ‘encrypted‘ string from the GUI instead.

5.4 Arduino IDE code

This Arduino sketch receives the encrypted output from the Enigma simulator (Python GUI) via serial communication and displays it on a 128x64 OLED screen using the Adafruit SSD1306 and GFX libraries. It waits for input from the serial buffer, appends each character to a message string, and dynamically updates the OLED screen.

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
```

These libraries handle I2C communication and rendering graphics on the OLED screen.

```
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64

#define OLED_RESET     -1
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire,
   OLED_RESET);
```

Initializes a 128x64 OLED display. The ‘OLED$_RESET$‘$pinisunusedandpassedas$‘ $-$ 1‘.

```
String incomingMessage = "";
```

This string accumulates incoming characters from the serial buffer.

```
void setup() {
  Serial.begin(9600);

  if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println(F("OLED allocation failed"));
    for (;;);
  }

  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 0);
  display.println("Waiting...");
  display.display();
}
```

The ‘setup()‘ function initializes serial communication and prepares the OLED screen. If the screen fails to initialize, the program halts.

```
void loop() {
  while (Serial.available() > 0) {
    char c = Serial.read();
    incomingMessage += c;

    if (incomingMessage.length() >= 128) {
      incomingMessage = "";
    }
```

```
33
34      display.clearDisplay();
35      display.setCursor(0, 0);
36      display.println(incomingMessage);
37      display.display();
38    }
39  }
```

In the main loop, the Arduino reads incoming serial characters, appends them to a string, and displays them on the OLED. If the string exceeds 128 characters, it's reset to prevent overflow.

5.5 Rotary Encoder Code

```
1  const int clkPin = 2;     // CLK pin of encoder
2  const int dtPin = 3;      // DT pin of encoder
3  const int swPin = 4;      // Button pin of encoder
4
5  // Rotor values
6  char rotors[3] = {'A', 'A', 'A'};
7  int currentRotor = 0;
8
9  // Encoder state
10 int lastClkState;
11 unsigned long lastButtonPress = 0;
12
13 void setup() {
14   pinMode(clkPin, INPUT);
15   pinMode(dtPin, INPUT);
16   pinMode(swPin, INPUT_PULLUP);
17
18   lastClkState = digitalRead(clkPin);
19
20   Serial.begin(9600);
21   Serial.println("Rotary Encoder Ready");
22   printRotors();
23 }
24
25 void loop() {
26   int newClkState = digitalRead(clkPin);
27   int dtState = digitalRead(dtPin);
28
29   // Handle rotation
30   if (newClkState != lastClkState) {
31     if (dtState != newClkState) {
32       // Clockwise - increment
33       rotors[currentRotor]++;
34       if (rotors[currentRotor] > 'Z') rotors[currentRotor] = 'A';
35     } else {
36       // Counter-clockwise - decrement
37       rotors[currentRotor]--;
```

```
      if (rotors[currentRotor] < 'A') rotors[currentRotor] = 'Z';
    }
    printRotors();
  }
  lastClkState = newClkState;

  // Handle button press to switch rotors
  if (digitalRead(swPin) == LOW) {
    if (millis() - lastButtonPress > 300) { // Debounce
      currentRotor = (currentRotor + 1) % 3;
      Serial.print("Switched to Rotor ");
      Serial.println(currentRotor + 1);
      lastButtonPress = millis();
    }
  }
}

void printRotors() {
  Serial.print("R1: ");
  Serial.print(rotors[0]);
  Serial.print("  R2: ");
  Serial.print(rotors[1]);
  Serial.print("  R3: ");
  Serial.println(rotors[2]);
}
```

Listing 1: Arduino code for setting rotor positions via rotary encoder

Functionality Summary

- Clockwise rotation increments the current rotor's letter.
- Counter-clockwise rotation decrements the letter.
- A button press switches the active rotor (R1 $\rightarrow$ R2 $\rightarrow$ R3).
- Rotor values are printed to Serial Monitor for visual feedback.

6 Working Mechanism

This section describes the working mechanism of a modern implementation of the Enigma Machine using Python and Arduino. The system includes a GUI for input, serial communication, rotor encryption logic, plugboard configuration, rotary encoder input, and LCD output.

System Components

- **PySimpleGUI:** Used for the graphical user interface where users input rotor settings, offsets, and plugboard connections.
- **PySerial:** Manages communication between the GUI (Python side) and Arduino (hardware).
- **Arduino:** Drives the 16x2 LCD display and reads input from a rotary encoder.
- **Rotary Encoder:** Manually changes rotor positions on the hardware side.
- **16x2 LCD:** Displays the encrypted message in real-time.
- **Plugboard (Steckerbrett):** A software-based letter mapping implemented in the GUI.

Workflow Description

The overall process is divided into the following phases:

1. **GUI Initialization:** The PySimpleGUI interface is initialized with fields for:
   - Rotor selection (e.g., Rotor I, II, III)
   - Rotor initial offset (A-Z)
   - Plugboard configuration (e.g., A $\leftrightarrow$ M, B $\leftrightarrow$ N)
   - Message input
2. **Plugboard Setup:** The plugboard swaps characters as the first and last step in encryption. The mapping is stored as a dictionary:

   ```
   plugboard = { 'A': 'M', 'M': 'A', 'B': 'N', 'N': 'B', ...}
   ```

3. **Rotor Configuration:** Each rotor is defined by a fixed internal wiring (substitution cipher) and a current position (offset). The rotor stepping mechanism mimics the actual Enigma behavior, rotating after each keystroke:

   ```
   EncryptedLetter = Rotor3(Rotor2(Rotor1(InputLetter + Offset)))
   ```

4. **Offset Input:** Users set the rotor's initial offsets using the GUI. These are sent via PySerial to the Arduino, which also tracks hardware-based changes via the rotary encoder.

5. **Rotary Encoder:** The encoder allows manual rotor movement. Each click changes the corresponding rotor's offset. The Arduino keeps track and can send updated offsets back to the PC via serial.
6. **Message Encryption:** The GUI encrypts the input message character-by-character as follows:
   (a) Apply plugboard swap
   (b) Pass through rotors (with offset handling)
   (c) Reflect using reflector logic
   (d) Return through rotors in reverse
   (e) Apply plugboard again
7. **Serial Communication:** The encrypted message is sent from Python to the Arduino using PySerial:

   ```
   serial.write("ENCRYPTED_TEXT_HERE")
   ```

8. **LCD Display:** The Arduino receives the encrypted message and displays it on the 16x2 LCD screen. The message is split across two lines if necessary.

Example Flow

1. User selects Rotor I, II, III with offsets A, B, C.
2. Plugboard mappings A $\leftrightarrow$ M, B $\leftrightarrow$ N are entered.
3. User types "HELLO" into the GUI.
4. The message is encrypted based on current rotor settings and plugboard.
5. Encrypted message is sent to Arduino.
6. Arduino displays it on LCD.
7. Rotary encoder is used to change offset of Rotor III during runtime.

Conclusion

This implementation of the Enigma Machine successfully blends historical cipher mechanics with modern hardware and software. The project demonstrates serial communication, GUI interaction, hardware integration, and encryption logic in a cohesive system.

7 Conclusion

The Enigma project has provided a valuable opportunity to explore the implementation of classical cryptographic concepts using modern embedded systems. Through the development process, we have constructed the core components of the encryption system, including the rotor, plugboard, and reflector logic, and validated their functionality through iterative testing and debugging.

As we approach the final phase of the project, several key tasks remain. These include fine-tuning the rotor and plugboard logic to ensure accurate encryption, integrating the rotary encoder for dynamic rotor offset adjustment, and implementing sound effects via DAC output to simulate typewriter keystrokes. Additionally, we aim to finalize the GUI, microcontroller code, and sound integration. Thorough testing and debugging of the entire system are still pending, followed by the submission of the final report and documentation.

This project has underscored the challenges and intricacies of building secure, real-time embedded systems. The remaining tasks will complete our effort to create a fully functional and interactive Enigma machine simulation, blending cryptographic logic with hardware-software integration.