1. Introducción

Este reporte detalla el diseño, desarrollo e implementación de un sistema de software en C++ capaz de decodificar el protocolo industrial "PRT-7". El desafío principal radica en que el protocolo no envía datos, sino **instrucciones** para ensamblar un mensaje oculto.

El sistema recibe dos tipos de tramas de un flujo serial (simulado):

- **Tramas de Carga (LOAD):** Contienen un fragmento de dato (un carácter).
- **Tramas de Mapeo (MAP):** Contienen instrucciones para reordenar el alfabeto de decodificación.

Para resolver este problema, se implementaron estructuras de datos manuales (sin STL) y un diseño polimórfico que permite procesar las diferentes tramas de forma dinámica. El proyecto fue compilado y probado en Ubuntu 24.04 utilizando CMake y el compilador G++.

---

2. Manual Técnico

2.1 Diseño y Arquitectura (POO Polimórfica)

La base del sistema es un diseño de Programación Orientada a Objetos (POO) que utiliza **polimorfismo** para manejar los distintos tipos de tramas. Esto permite que el bucle principal del programa (`main.cpp`) procese cualquier tipo de trama futura (ej. `TramaEncrypt`, `TramaPing`, etc.) sin necesidad de modificar el código existente.

Se definió una jerarquía de clases:

- **`TramaBase` (Clase Abstracta):** Es la interfaz común. Define un método virtual puro `procesar()` que todas las tramas concretas deben implementar. Crucialmente, incluye un **destructor virtual** (`virtual ~TramaBase()`) para garantizar la correcta liberación de memoria al usar punteros de la clase base.
- **`TramaLoad` (Clase Derivada):** Hereda de `TramaBase`. Almacena un `char` como *payload*. Su lógica `procesar()` consiste en pedir al rotor el mapeo del carácter y luego insertarlo en la lista de carga.
- **`TramaMap` (Clase Derivada):** Hereda de `TramaBase`. Almacena un `int` como *payload* (la rotación). Su lógica `procesar()` es simple: le ordena al rotor que rote N posiciones.

2.2 Estructuras de Datos Implementadas

De acuerdo a los requisitos, se prohibió el uso de la STL (`std::list`, `std::vector`). Por lo tanto, todas las estructuras de datos se implementaron manualmente.

ListaDeCarga (Lista Doblemente Enlazada)

Esta estructura almacena el mensaje final decodificado. Se eligió una lista doblemente enlazada por la eficiencia al insertar al final (operación O(1) si se mantiene un puntero a la cola).

- **Nodos:** `NodoCarga` (almacena `char dato`, `NodoCarga* previo`, `NodoCarga* siguiente`).
- **Métodos Clave:**
  - `insertarAlFinal(char dato)`: Añade un nuevo nodo al final de la lista, actualizando el puntero `cola`.
  - `imprimirMensaje()`: Recorre la lista desde la `cabeza` hasta el final, imprimiendo el `dato` de cada nodo.
  - `~ListaDeCarga()`: El destructor recorre la lista y libera la memoria de cada `NodoCarga` uno por uno para evitar fugas de memoria.

RotorDeMapeo (Lista Circular Doblemente Enlazada)

Esta es la estructura más compleja; actúa como el "disco de cifrado" dinámico. Es una lista circular que contiene el alfabeto (A-Z).

- **Nodos:** `NodoRotor` (almacena `char dato`, `NodoRotor* previo`, `NodoRotor* siguiente`).
- **Constructor:** El constructor `RotorDeMapeo()` se encarga de crear los 26 nodos (A-Z), enlazarlos y finalmente "cerrar el círculo" conectando el nodo 'Z' con el nodo 'A'.
- **Puntero `cabeza`:** Este puntero es el estado del rotor. Indica la posición 'cero' actual (a qué carácter se mapea 'A').
- **Métodos Clave:**
  - `rotar(int N)`: Es una operación O(N) muy eficiente. Simplemente mueve el puntero `cabeza` N posiciones (hacia `siguiente` si N es positivo, o `previo` si N es negativo).
  - `getMapeo(char in)`: Calcula el carácter decodificado. Primero, calcula el *offset* del carácter de entrada (ej. 'A'=0, 'B'=1, 'C'=2). Luego, avanza ese *offset* de posiciones desde el puntero `cabeza` actual para encontrar el carácter mapeado.

2.3 Componentes del Sistema (Archivos)

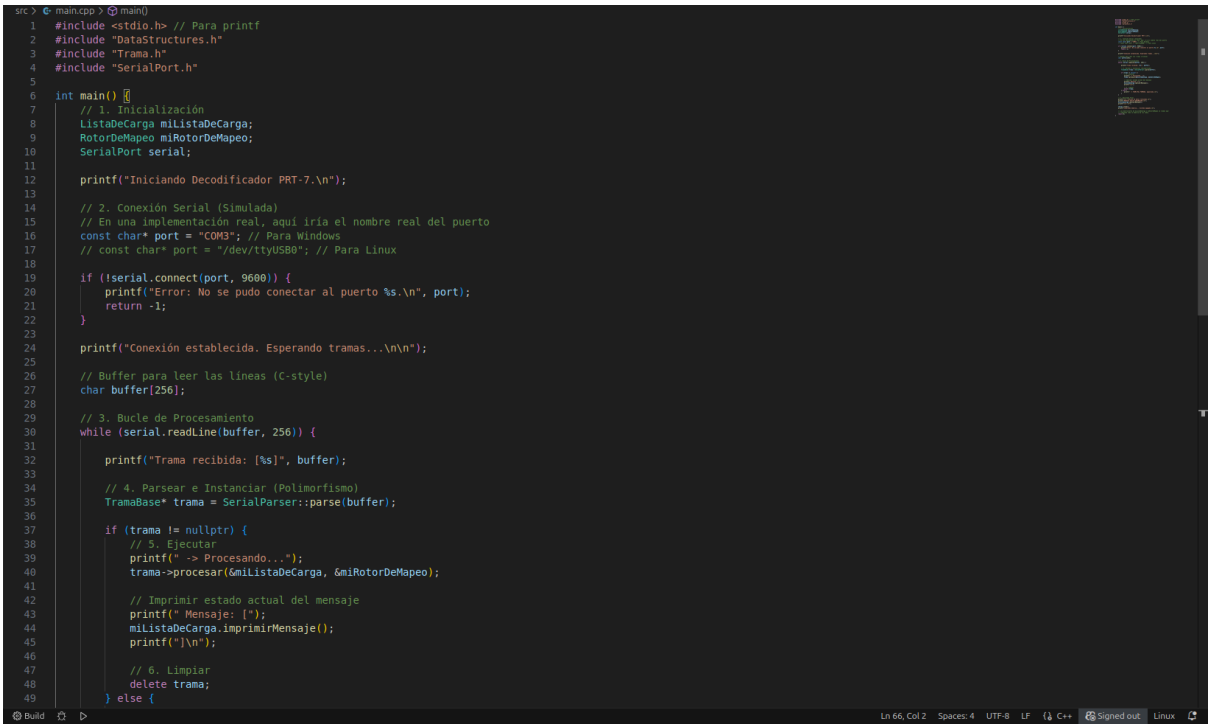
El proyecto se organizó en una estructura modular estándar de CMake:

- **`CMakeLists.txt`:** Define el proyecto, enlaza los ejecutables y las carpetas `include`/`src`. También incluye un objetivo `doc` para generar la documentación Doxygen.
- **`include/DataStructures.h`:** Define las interfaces de `ListaDeCarga` y `RotorDeMapeo`.
- **`include/Trama.h`:** Define la jerarquía de clases (`TramaBase`, `TramaLoad`, `TramaMap`) y el *namespace* `SerialParser`.
- **`include/SerialPort.h`:** Define la interfaz de la clase `SerialPort` (la versión simulada).
- **`src/DataStructures.cpp`:** Contiene la implementación (la lógica) de `ListaDeCarga` y `RotorDeMapeo`.
- **`src/Trama.cpp`:** Contiene la implementación de los métodos `procesar()` de cada trama y la lógica de `SerialParser::parse()` para convertir texto (ej. "L,A") en objetos.
- **`src/SerialPort.cpp`:** Implementación del simulador de puerto serial que entrega las tramas de ejemplo con un retardo de 1 segundo.
- **`src/main.cpp`:** El orquestador. Inicializa las estructuras, se "conecta" al puerto, lee líneas en un bucle, usa el `SerialParser` para crear objetos `TramaBase*` en el *heap* (`new`), ejecuta `trama->procesar()`, y finalmente libera la memoria (`delete trama`).

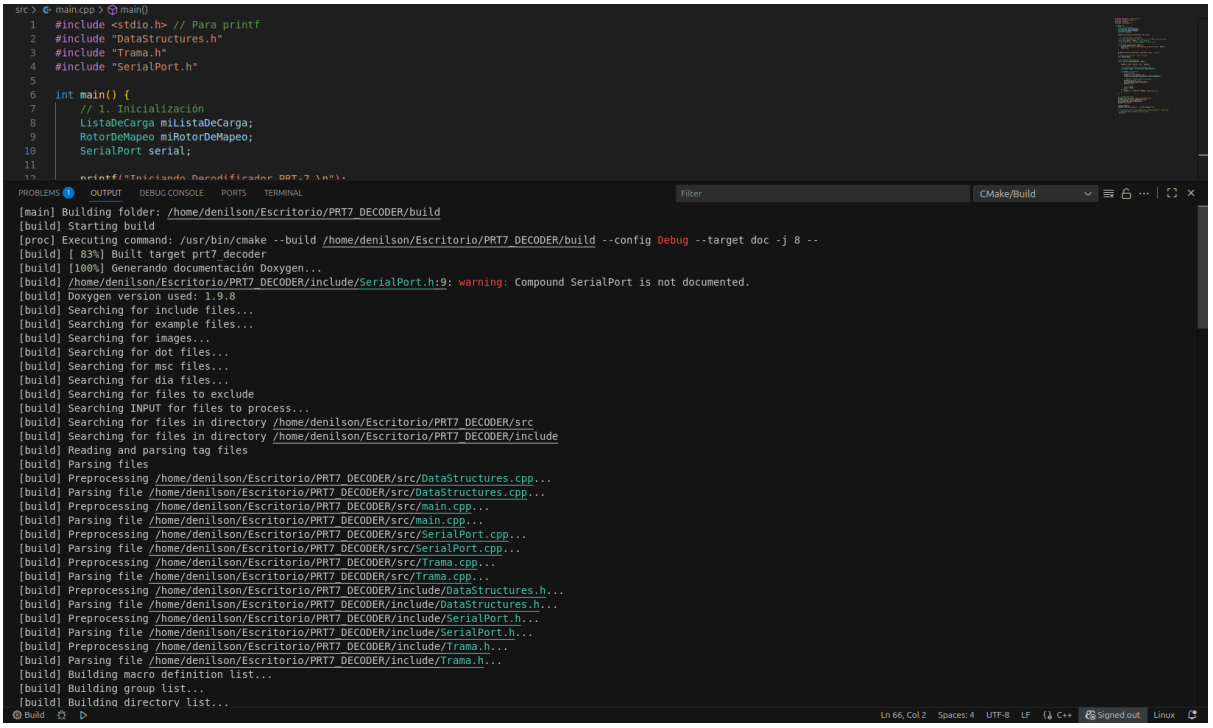
3. Pantallazos de la Implementación

A continuación, se presentan capturas de pantalla que evidencian el funcionamiento del sistema.

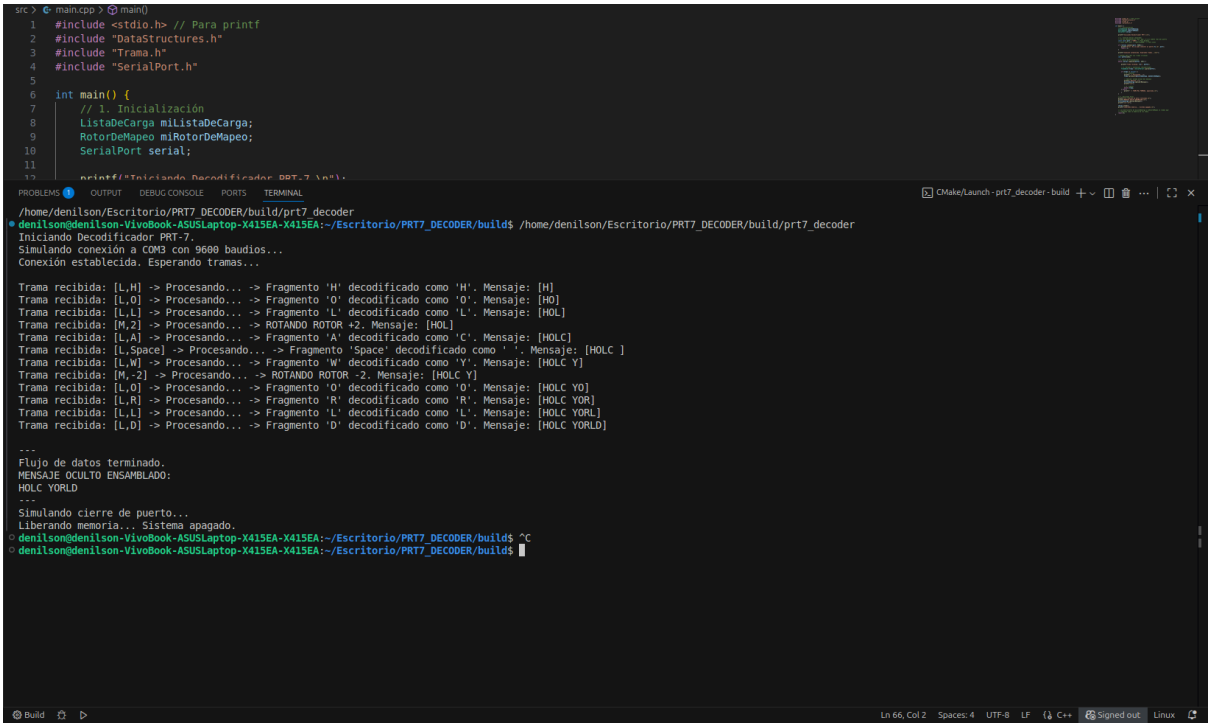
1. Código Fuente (Ej. `main.cpp` y la jerarquía `Trama.h`)



2. Compilación Exitosa (Salida de `[ Build ]`)



3. Ejecución del Programa (Salida de `[ Run ]`)



4. Documentación Doxygen Generada