UNIVERSIDADE FEDERAL DE JUIZ DE FORA
FACULDADE DE ENGENHARIA
ENGENHARIA ELÉTRICA - ROBÓTICA E AUTOMAÇÃO INDUSTRIAL

Matheus Taninho Reinh

Desenvolvimento de um Painel de Controle Local com Comunicação via Protocolo Genisys

Juiz de Fora

2025

Matheus Taninho Reinh

Desenvolvimento de um Painel de Controle Local com Comunicação via Protocolo Genisys

Trabalho de conclusão de curso apresentado ao Departamento de Energia Elétrica da Universidade Federal de Juiz de Fora como requisito para aprovação na disciplina - Trabalho de Final de Curso.

Orientador: Prof. Dr. Guilherme Márcio Soares

Juiz de Fora

2025

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Taninho Reinh, Matheus.
Desenvolvimento de um Painel de Controle Local com Comunicação via Protocolo Genisys / Matheus Taninho Reinh. – 2025.
62 f. : il.

Orientador: Guilherme Márcio Soares
Trabalho de Conclusão de Curso de Graduação – Universidade Federal de Juiz de Fora, Faculdade de Engenharia. Engenharia Elétrica - Robótica e Automação Industrial, 2025.

1. IHM. 2. *Genisys*. 3. Python, *Kivy* I. Soares, Guilherme M., orient. II. Título.

Matheus Taninho Reinh

Desenvolvimento de um Painel de Controle Local com Comunicação via Protocolo Genisys

Trabalho de conclusão de curso apresentado ao Departamento de Energia Elétrica da Universidade Federal de Juiz de Fora como requisito para aprovação na disciplina - Trabalho de Final de Curso.

BANCA EXAMINADORA

Prof. Dr. Guilherme Márcio Soares - Orientador
Universidade Federal de Juiz de Fora

Prof. Dr. Exuperry Barros Costa
Universidade Federal de Juiz de Fora

Eng. Thiago Modesto Oliveira
Wabtec Corporation

AGRADECIMENTOS

Soli deo Gloria. Porque dEle, e por meio dEle, e para Ele são todas as coisas.

Gostaria de expressar minha mais profunda gratidão a Deus. Sem Ele eu nada sou. O Senhor de todo o universo, que por sua superabundante graça e infinito amor, tornou o miserável pecador que sou, em seu filho, e com sua misericórdia e bondade leva-me a conquistas, as quais jamais sonhei alcançar.

Quero agradecer à minha família. Meus irmãos, que sempre foram inspiração e bons exemplos para mim, e especialmente aos meus pais, que me moldaram e com muito amor e carinho pavimentaram o caminho que trilhei até aqui.

À minha esposa, minha gratidão por seu amor, sua paciência e apoio, sempre me motivando, sendo minha base para não desistir no momentos de dificuldade e minha alegria para seguir a jornada da vida a cada dia.

Aos meus professores, que ao longo do curso, cada um com a sua particularidade, compartilharam seus conhecimentos, experiências e orientações, muito obrigado.

A eletricidade é a alma do universo moderno.

George Westinghouse

RESUMO

Este trabalho apresenta o desenvolvimento de uma Interface Humano-Máquina para sistemas ferroviários, substituindo uma solução legado baseada em Adobe Flash. O objetivo central foi criar um sistema moderno que se comunica com o controlador lógico programável, ElectroLogIXS, via protocolo Genisys, utilizado em sistemas de sinalização ferroviária. A metodologia incluiu a decodificação do protocolo Genisys, desenvolvimento de um *driver* em Python para a comunicação serial, e a implementação de uma IHM utilizando o *framework* Kivy, permitindo visualização em tempo real do estado de dispositivos de campo (sinais, circuitos de via, máquinas de chave) e envio de comando de controle. A solução foi validada através do simulador *Signal Application Testing System* (SATS), demonstrando precisão na interpretação de mensagens de indicação e eficácia no envio de comandos. Resultados destacam a superação das limitações do sistema legado, como a falta de personalização, dificuldade de instalação e depêndencia de tecnologias obsoletas, oferecendo uma alternativa escalável, de código aberto e com total acesso ao *backend*. Conclui-se que a integração entre o *driver* Genisis e a IHM em Kivy representa um avanço significativo, com potencial para a aplicação em cenários reais e futuras expansões, como suporte para TCP/IP e gestão de *logs* integrada.

Palavras-chave: IHM, Genisys, Python, ElectroLogIXS, Kivy.

ABSTRACT

This work presents the development of a Human-Machine Interface (HMI) for railway systems, replacing a legacy solution based on Adobe Flash. The main objective was to create a modern system that communicates with the programmable logic controller, ElectroLogIXS, via the Genisys protocol, used in railway signaling and control systems. The methodology included decoding the Genisys protocol, developing a Python driver for serial communication, and implementing an HMI using the Kivy framework, enabling real-time visualization of field device status (signals, track circuits, switch machines) and transmission of control commands. The solution was validated through the Signal Application Testing System (SATS) simulator, demonstrating accuracy in interpreting status messages and effectiveness in command transmission. Results highlight the overcoming of legacy system limitations, such as lack of customization, installation challenges, and dependency on obsolete technologies, offering a scalable, open-source alternative with full access to the backend. It is concluded that the integration between the Genisys driver and the Kivy-based HMI represents a significant advancement for railway automation, with potential for real-world application and future expansions, such as TCP/IP support and integrated log management.

Keywords: HIM, Genisys, Python, ElectroLogIXS, Kivy.

LISTA DE ILUSTRAÇÕES

Figura 1 – Aba *Location Designer* do SATS 20
Figura 2 – Comms SATS 20
Figura 3 – Sistema Completo 21
Figura 4 – Fluxograma do código 23
Figura 5 – *Log* do SATS 25
Figura 6 – Mudança de *Status* 25
Figura 7 – Sexto Octeto do Mapa de *Bits* 26
Figura 8 – Estrutura do *frame* Genisys 27
Figura 9 – Mensagem Bruta no Monitor Serial 27
Figura 10 – Exemplo de LED 33
Figura 11 – Exemplo de Sinal 34
Figura 12 – Exemplo de Circuito de Via Ocupado 37
Figura 13 – Exemplo de Circuito de Via Livre 39
Figura 14 – Exemplo de Máquina de Chave 39
Figura 15 – Exemplo de Botão 42
Figura 16 – Estado Inicial do SATS 50
Figura 17 – Estado Inicial do PCL 51
Figura 18 – Segundo Estado do SATS 51
Figura 19 – Segundo Estado do PCL 52
Figura 20 – Controle recebido pelo SATS 53
Figura 21 – Primeira Interface PCL Legado 54
Figura 22 – Segunda Interface PCL Legado 54
Figura 23 – Terceira Interface PCL Legado 55
Figura 24 – Configurações de conexão PCL Legado 56
Figura 25 – Monitor PCL Legado 56
Figura 26 – Interface de Conexão PCL 57
Figura 27 – Interface de Conexão PCL 58

LISTA DE TABELAS

Tabela 1 – Tabela Comparativa entre Sistema Desenvolvido e Sistema Legado 48
Tabela 2 – Tabela Comparativa de Desempenho e Usabilidade 59

LISTA DE ABREVIATURAS E SIGLAS

ANTT	*Agencia Nacional de Transportes Terrestres*
ATCS	*Advanced Train Control System*
CCO	*Centro de Controle Operacional*
CLP	*Controlador Lógico Programável*
CRC	*Cyclic Redundancy Check*
ELIX	*ElectroLogIXS*
IA	*Inteligência Artificial*
ICSNPP	*Industrial Control Systems Network Protocol Parsers*
IHM	*Interface Humano-Máquina*
MCH	*Máquina de Chave Elétrica*
PCL	*Painel de Controle Local*
PTC	*Positive Train COntrol*
SATS	*Signal Application Testing System*
TCP/IP	*Transmission Control Protocol/Internet Protocol*
UDP/IP	*User Dataprogram Protocol/Internet Protocol*

SUMÁRIO

1	**Introdução**	**12**
1.1	Contextualização	12
1.2	Problemas e Dores do Sistema Legado	13
1.3	Motivação	13
1.4	Objetivos Gerais	14
1.5	Objetivos Específicos	14
1.6	Organização do Trabalho	14
2	**Revisão Bibliográfica e Arcabouço Teórico**	**15**
2.1	Revisão Bibliográfica	15
2.2	Arcabouço Teórico	16
2.2.1	Python	17
2.2.2	*Framework* Kivy	17
2.2.3	Simulador de CLP - SATS	19
3	**Desenvolvimento**	**21**
3.1	Fluxograma do Código	22
3.1.1	Inicialização e Conexão	23
3.1.2	Processamento de Dados e Atualização de Estado	24
3.1.3	Interface de Supervisão e Interação	24
3.2	Desenvolvimento do *Driver* Genisys	24
3.2.1	Estrutura do *Frame* Genisys	26
3.3	Aplicação do *Driver* Genisys no Código	28
3.3.1	Inicialização e Ciclo de Vida do *Driver*	28
3.3.2	Estabelecimento de Conexão	29
3.3.3	*Thread* de Comunicação Serial	29
3.3.3.1	Política de *Polling*	30
3.3.4	Processamento de Indicações	30
3.3.5	Sincronização com a Interface	31
3.4	Desenvolvimento da Interface Humano-Máquina	31
3.4.1	LED's	33
3.4.2	Sinais	34
3.4.3	Circuito de Via	37
3.4.4	Máquina de Chave	39
3.4.5	Botões	42
3.5	Conexão da IHM com o *Driver*	44
3.5.1	Envio de Mensagens de Controle	44
3.5.2	Interpretação das Mensagens de Indicação	47
3.6	Comparativo Entre a Interface Desenvolvida e o Sistema Legado	48

3.6.1 Desenvolvimento da Interface Gráfica 49
3.6.2 Suporte 49
3.6.3 Acesso ao *Backend* 49
3.6.4 Customização 49
4 Resultados 50
4.1 Resultados da Decodificação de Mensagens 50
4.2 Avaliação da Interface Gráfica 53
4.3 Desempenho e Usabilidade. 58
5 Conclusão e Propostas Para Trabalhos Futuros 60
5.1 Contribuições do Trabalho 60
5.2 Propostas Para Trabalhos Futuros 60
5.3 Considerações Finais 61
REFERÊNCIAS 62

1 Introdução

1.1 Contextualização

O transporte ferroviário é essencial para o avanço e desenvolvimento de um país de proporções continentais. Em países como Estados Unidos e China existe uma malha ferroviária que cruza toda sua a extensão territorial. O Brasil de maneira modesta em comparação aos países citados anteriormente também possui investimentos no setor.

A ferrovia, antes movida por energia termoelétrica com locomotivas que dependiam da queima do carvão, possui hoje locomotivas elétricas, uma demonstração dos constantes avanços tecnológicos presentes nos sistemas ferroviários. Esses avanços são essenciais para que haja aumento de produtividade sem que a segurança seja negligenciada.

Contemporaneamente, as empresas que possuem a concessão das malhas ferroviárias do Brasil negociam com a Agência Nacional de Transportes Terrestres (ANTT) alguns marcos de avanços que devem ser realizados pelas empresas no tempo em que a ferrovia estiver sob sua responsabilidade. Dessa maneira as concessionárias podem usufruir da infraestrutura presente, mas devem investir constantemente em melhorias, evitando assim que aconteça um sucateamento da malha do país.

Diante desse cenário, sistemas com automação e controle se tornam muito presentes no setor, pois garantem produtividade com segurança, consequentemente aumentam o lucro para as empresas, evitando impactos negativos com acidentes. Dentre esses se destaca o PTC (do inglês, *Positive Train Control*), que segundo [1], é feita para evitar colisões entre trens, descarrilamentos causados por velocidade excessiva não autorizada e avanços de trens para regiões que estão em manutenção. Uma aplicação de PTC possui no seu sistema mais básico ao menos os seguintes elementos:

- Sistema de Controle de Locomotivas: Equipamentos instalados nas locomotivas que recebem e transmitem dados em tempo real, possibilitando o controle automático do trem em emergências.
- Sistema de Sinalização e Comunicação: Infraestrutura ao longo da via férrea que comunica informações críticas para a locomotiva ou Centro de Controle Operacional
- Centro de Controle de Operações: Uma central que gerencia a monitora o tráfego ferroviário.
- *Softwares* de Gestão e Análise: Programas que processam informações de sensores e do sistema de comunicação e tomam decisões automáticas, visando a prevenção de acidentes.

Portanto, essas tecnologias não apenas transformam a maneira de gerenciar frotas veiculares, mas também abrem portas para uma era de transporte mais inteligente, responsiva e sustentável. A evolução contínua dessas soluções promete uma revolução no setor ferroviário e logístico, trazendo consigo oportunidades para aumentar a competitividade e impulsionar o crescimento econômico.

1.2 Problemas e Dores do Sistema Legado

Entre os sistemas e equipamentos de campo que tornam possível a implementação de soluções robustas para a ferrovia está o Painel de Controle Local (PCL), individual e personalizado para cada configuração de pátio ferroviário. O painel apresenta o *status* atual da via e dos sensores monitorados pelo ElectroLogIXS (ELIX) e permite que, em determinadas condições, comandos sejam enviados ao equipamento.

O Sistema legado é desenvolvido através do Adobe Flash e possui duas aplicações que atuam em conjunto. A primeira é utilizada para criar a interface gráfica, associando elementos gráficos a determinados endereços do mapa de *bits* do *software* instalado no ELIX, resultando em um arquivo com extensão `.swf` que por sua vez é utilizado na segunda aplicação, que finalmente realiza a comunicação com o CLP via protocolo Genisys.

A arquitetura de desenvolvimento atual do PCL, baseada em Flash, não permite acesso ao *backend* da aplicação que estabelece comunicação via protocolo Genisys, permitindo pouca ou nenhuma personalização para atender casos que fujam do que já foi desenvolvido anteriormente. Além disso, a instalação do programa é muito custosa, por necessitar de diversos arquivos e subaplicações em pastas específicas do computador para rodar o programa principal. Esses problemas se agravaram com a descontinuação da ferramenta no ano de 2020, dificultando a criação de novas Interfaces Humano-Máquina personalizadas para cada configuração de pátio ferroviário.

1.3 Motivação

Ao ter o primeiro contato com o sistema utilizado para criar os Painéis de Controle Local atuais, muitas dificuldades se apresentaram desde a instalação da aplicação de desenvolvimento de interfaces até ao programa que de fato realiza a comunicação com o CLP.

O PCL é utilizado em momentos críticos, que geralmente precisam de agilidade e garantia de sucesso, por ser um *software* de utilização em campo, em situações que podem impactar a operação em caso de falhas.

Diante desse cenário, surgiu o desafio de desenvolver um sistema que substituísse o atual, utilizando uma nova linguagem de programação, e trazendo mais confiabilidade e garantia para quem precisar utilizar o programa.

Vale ressaltar que como a solução atual não permite acesso ao *backend*, o desenvolvimento de um *driver* de comunicação via protocolo Genisys que é o protocolo utilizado para troca de mensagens entre o sistema legado e o CLP em questão, se torna essencial para o sucesso do trabalho.

1.4 Objetivos Gerais

O objetivo geral deste trabalho é desenvolver um *driver* de comunicação para o protocolo Genisys e uma Interface Humano-Máquina (IHM) para a exibição das indicações do sistema de sinalização ferroviária e envio de comandos de controle para o mesmo, a fim de substituir o sistema existente.

1.5 Objetivos Específicos

Para atingir o objetivo geral deste projeto, foi necessário:

- Analisar e decodificar mensagens do protocolo Genisys;
- Desenvolver um *driver* de comunicação para o protocolo;
- Criar uma IHM utilizando o *framework* Kivy;
- Comparar a solução desenvolvida com o sistema legado;

1.6 Organização do Trabalho

Este documento está estruturado em cinco capítulos fundamentais para apresentação coerente do desenvolvimento e validação da solução proposta. No Capítulo 2, são discutidos os conceitos teóricos relacionados ao protocolo Genisys, tecnologias de IHMs e sistemas ferroviários, além de uma análise crítica de trabalhos correlatos que fundamentaram as escolhas técnicas deste projeto. O Capítulo 3 detalha a implementação prática, abordando a decodificação do protocolo Genisys, arquitetura do *driver* de comunicação, e o processo de construção da interface gráfica com o *framework* Kivy.

No Capítulo 4, são apresentados os testes de validação funcional com o simulador *Signal Application Testing System* (SATS), análise comparativa com o sistema legado, e avaliação de desempenho da solução desenvolvida. Por fim, o Capítulo 5 sintetiza as contribuições do trabalho, discute limitações identificadas, e propõe direções para evolução da plataforma.

2 Revisão Bibliográfica e Arcabouço Teórico

2.1 Revisão Bibliográfica

Esta seção apresenta uma breve revisão de alguns trabalhos e documentos que foram utilizados como referência para o desenvolvimento deste projeto. Esses resumos visam oferecer uma visão geral dos principais pontos abordados em cada trabalho, destacando suas contribuições relevantes para o tema em estudo.

Em [2], Bruce Keeler apresenta um *dissector*[1], para o protocolo Genisys, desenvolvido para a ferramenta Wireshark. O protocolo Genisys, utilizado no controle de sistemas de sinalização e intertravamentos ferroviários, é descrito em detalhes, incluindo sua estrutura de mensagens, mecanismos de escape de dados e cálculo de CRC-16. O autor destaca que o Genisys foi originalmente projetado para comunicação serial, mas é frequentemente transportado sobre TCP/IP em implementações modernas. O *dissector* desenvolvido por Keeler permite a análise de pacotes Genisys, identificando mensagens como dados de indicação, controle, e reconhecimento, além de validar a integridade dos dados através do CRC. Esse trabalho é fundamental para a compreensão do protocolo Genisys, fornecendo uma base técnica para o desenvolvimento de *drivers* de comunicação e sistemas de supervisão baseados nesse protocolo, como proposto neste trabalho.

Em [3], o projeto *Industrial Control Systems Network Protocol Parsers* (ICSNPP) apresenta um *plugin* para o Zeek, desenvolvido em Spicy, capaz de analisar e registrar mensagens do protocolo Genisys transportadas sobre TCP/IP. O *plugin* ICSNPP-Genisys foi desenvolvido com base na engenharia reversa de um *packet capture* de tráfego Genisys e em referências a um *dissector* não oficial proposto para o Wireshark. Ele gera um arquivo de *log* (genisys.log) que captura detalhes como tipo de mensagem, direção (requisição/resposta), CRC transmitido e calculado, e pares de endereço-dado do *payload*. Esse trabalho é relevante para este projeto, pois fornece uma implementação moderna e validada do parser Genisys, auxiliando na compreensão e no desenvolvimento de soluções de comunicação baseadas nesse protocolo.

Em [4], o manual técnico da empresa Union Switch & Signal (US&S) detalha especificações do protocolo Genisys Serial, utilizado para controle e supervisão de sistemas ferroviários. O protocolo, de natureza binária e orientado a *bytes*, opera em modo mestre-escravo com mensagens estruturadas em cabeçalho, endereço de conteúdo, dados opcionais, *checksum* CRC-16 e terminador. A comunicação é baseada em caracteres de controle reservados, como o F6 para terminador, e mecanismos de escape para preservar a integridade de *bytes* críticos no *payload*. O documento descreve formatos específicos para mensagens mestre-escravo, incluindo um *header* para cada tipo de mensagem. Além

[1] *Dissector* é um módulo usado pelo Wireshark para analisar protocolos campo por campo, permitindo filtros em critérios específicos para análise do protocolo.

disso, aborda a operação dos *drivers* do protocolo, com lógicas de retransmissão, timeouts e tratamento de falhas, bem como a configuração de *bytes* de controle. Esse trabalho é essencial para o projeto atual, pois fornece a base normativa para a implementação de *drivers* de comunicação compatíveis com o Genisys, validando estruturas de mensagens, cálculos de CRC-16 e fluxos de interação mestre-escravo, críticos para o desenvolvimento de sistemas de supervisão confiáveis em ambientes ferroviários.

Em [5], Sergey N. Verzynov e colaboradores propõem uma arquitetura *cross-platform* para o componente de *software* de um localizador de cabos, utilizando a biblioteca Kivy para desenvolver uma interface gráfica unificada em dispositivos móveis e computadores de mesa, os autores criaram uma arquitetura que separa o código dependente da plataforma, como *drivers* de *hardware* e configurações específicas do sistema operacional, do código universal, lógica de processamento e visualização de dados. Ferramentas como *Buildozer* e *CMake* foram empregadas para empacotar o aplicativo em Android, enquanto o *Docker* garantiu a reprodutibilidade do ambiente de desenvolvimento. O componente foi testado com sucesso em dispositivos Android, integrando sensores como GPS e bússola, e permitindo a visualização de rotas de cabos em mapas digitais. O trabalho demonstra a viabilidade de implementação do *software* para iOS e Windows 10 com adaptações mínimas, mantendo a mesma base de código inicial em Python.

A revisão bibliográfica demonstra a viabilidade técnica tanto do desenvolvimento de um *driver* para o protocolo Genisys quanto da construção de uma IHM utilizando o *framework* Kivy. Os trabalhos de [2], [3] e [4] fornecem bases para a implementação do *driver*, detalhando a estrutura de mensagens, mecanismos de escape, cálculo de CRC-16 e fluxos mestre/escravo. A especificação técnica da US&S [4] e a existência de *parsers* validados, como o *plugin* ICSNPP-Genisys, comprovam que o protocolo é replicável em sistemas modernos, mesmo em cenários baseados em TCP/IP.

Quanto à IHM, o estudo de [5] evidencia a capacidade do Kivy em suportar o desenvolvimento de interfaces *cross-platform* robustas, com integração de funcionalidades complexas e adaptações mínimas entre sistemas operacionais. A experiência bem-sucedida na integração de *hardware* e exibição de dados em tempo real confirma que o *framework* é uma solução eficaz para a IHM, garantindo portabilidade e atualizações dinâmicas de estados.

2.2 Arcabouço Teórico

Esta seção aborda as principais tecnologias utilizadas no desenvolvimento da solução proposta.

2.2.1 Python

Python é uma linguagem de programação de alto nível que tem ganhado destaque em diversas áreas da tecnologia devido à sua simplicidade e versatilidade. No contexto do desenvolvimento de *drivers* de comunicação, Python oferece uma série de vantagens que facilitam a implementação e manutenção desses componentes essenciais. A sintaxe clara e concisa da linguagem permite uma escrita de códigos mais legíveis e menos propensos a erros, o que é crucial para a confiabilidade dos *drivers* de comunicação. Além disso, a vasta coleção de bibliotecas e *frameworks* disponíveis, como pySerial e *socket*, proporciona ferramentas poderosas para a criação de soluções eficientes e robustas.

No desenvolvimento de Interfaces Humano-Máquina (IHM), a linguagem também se destaca por sua capacidade de acelerar o processo de criação de interfaces gráficas de usuário. *Frameworks* como Tkinter, PyQt e Kivy facilitam a construção de IHMs. A flexibilidade da linguagem facilita a integração das IHMs com outros sistemas e dispositivos, tornando-as mais versáteis e funcionais. Além disso, a capacidade de Python de suportar *multithreading* garante que as interfaces permaneçam responsivas, mesmo quando executam tarefas complexas em segundo plano.

No desenvolvimento de *drivers* de comunicação e IHMs, uma linguagem multiplataforma, como o Python, permite que os desenvolvedores criem soluções que podem ser executadas em diferentes sistemas operacionais sem a necessidade de grandes modificações. Isso aumenta a flexibilidade dos projetos, e também reduz o tempo e os custos associados à adaptação do *software* para diferentes ambientes. A comunidade ativa de desenvolvedores Python contribui para um suporte contínuo, atualizações frequentes e uma vasta quantidade de recursos e documentação.

O Python se apresenta como uma boa escolha para o desenvolvimento de *drivers* de comunicação aliado a Interfaces Humano-Máquina. A simplicidade e legibilidade da linguagem, combinadas com a ampla disponibilidade de bibliotecas e *frameworks*, facilitam a criação de soluções eficientes e robustas. A portabilidade e a capacidade de integração com outros sistemas aumentam ainda mais a aplicabilidade do Python nesses contextos.

2.2.2 *Framework* Kivy

Kivy é um *framework* de código aberto para o desenvolvimento de aplicações multimídia e interfaces gráficas em Python. Ele é especialmente projetado para criar aplicativos que funcionam em várias plataformas, incluindo Windows, macOS, Linux, iOS e Android. Uma das principais características do Kivy é sua capacidade de suportar *multitouch*, o que o torna ideal para o desenvolvimento de aplicativos modernos e interativos. Além disso, Kivy utiliza um mecanismo de renderização baseado em OpenGL ES 2, garantindo que as interfaces gráficas sejam rápidas e responsivas, mesmo em dispositivos com recursos limitados.

A natureza multiplataforma do *framework* permite que os desenvolvedores escrevam código uma vez e o executem em diferentes sistemas operacionais, economizando tempo e esforço. O Kivy oferece uma vasta coleção de *widgets* prontos para uso, que podem ser facilmente personalizados para atender às necessidades específicas do projeto. Isso acelera o processo de desenvolvimento e reduz a complexidade da criação de interfaces gráficas.

O *framework* estrutura sua interface gráfica por meio de um sistema hierárquico de *widgets*, que permite a composição modular de elementos visuais predefinidos, como botões e *layouts* ou elementos personalizados. A linguagem declarativa KV introduz uma camada de abstração para definição de interfaces, separando lógica de programação em Python da estrutura da interface. Através de sintaxe específica, estabelecem-se relações hierárquicas entre *widgets*, estilos visuais e regras de posicionamento.

Entre os diversos *widgets* oferecidos pelo Kivy, estão pontuados alguns que serão utilizados neste trabalho:

- *FloatLayout*: Permite posicionar elementos de interface livremente na tela, utilizando coordenadas relativas ou absolutas. Isso é ideal para designs personalizados e sobreposições precisas. O *FloatLayout* é especialmente útil quando se deseja criar interfaces flexíveis que se adaptem a diferentes tamanhos de tela e resoluções. Ele permite que os desenvolvedores controlem a posição exata dos *widgets*, o que é essencial para aplicações que exigem um *layout* altamente customizado;

- *BoxLayout*: Organiza os componentes em sequência vertical ou horizontal, distribuindo o espaço automaticamente entre eles. Isso simplifica a criação de estruturas alinhadas e adaptáveis a diferentes tamanhos de tela. O *BoxLayout* é frequentemente utilizado para criar *layouts* que precisam ser redimensionados de forma proporcional. Ele é ideal para criar interfaces que se ajustam automaticamente ao redimensionamento da janela, mantendo a proporção dos *widgets* internos;

- *Image*: Essencial para carregar e exibir imagens com suporte a formatos variados, como PNG, JPEG, e GIF. O *widget Image* pode ser utilizado para mostrar logotipos, ícones, ou qualquer outro tipo de imagem gráfica. As imagens carregadas podem ser configuradas para diferentes tamanho e posições, podendo ser integrada a outros *widgets* de *layout*;

- *Label*: Oferece flexibilidade na exibição de textos, com opções de formatação e estilo. O *Label* é utilizado para mostrar informações textuais ao usuário, como títulos, descrições, ou mensagens de *status*. Ele suporta a personalização de fontes, cores, e alinhamento, permitindo que os desenvolvedores criem interfaces textuais atraentes. O *Label* também pode ser utilizado em conjunto com outros *widgets* para criar interfaces informativas e visualmente agradáveis;

- *Button*: Fundamental para interatividade, permitindo a associação de ações a eventos como cliques ou toques. O *Button* pode ser configurado para executar funções específicas quando acionado, tornando-se um elemento crucial para a navegação e interação do usuário. Ele pode ser estilizado com diferentes cores, tamanhos, e ícones para se adequar ao design da aplicação. Além disso, o *Button* pode ser utilizado em conjunto com o *BoxLayout* para criar painéis de controle interativos.

Esses *widgets*, combinados com a capacidade de personalização via propriedades de estilo e comportamentos, aceleram o desenvolvimento de interfaces complexas e garantem uma experiência consistente em diferentes dispositivos, reforçando a eficiência do Kivy em projetos de IHM.

A escolha do Kivy como *framework* para o desenvolvimento de uma Interface Humano-Máquina (IHM) é motivada por sua flexibilidade e eficiência. A capacidade de Kivy de criar interfaces intuitivas e interativas, torna-o ideal para aplicações que exigem uma experiência de usuário rica e consistente. Além disso, a compatibilidade multiplataforma de Kivy garante que a IHM desenvolvida possa ser utilizada em diversos dispositivos, ampliando seu alcance e aplicabilidade. Por fim, a possibilidade de fechar a solução e comercializá-la sem custos adicionais de licença, diferente do que ocorre com o PyQT e o TKinter, se tornou um fator determinante na escolha do *framework*.

2.2.3 Simulador de CLP - SATS

O *Signal Application Testing System* (SATS) é um simulador avançado desenvolvido pela Princeton Consultants para o teste e a simulação de sistemas de sinalização ferroviária. Esse *software* é amplamente utilizado por engenheiros, para melhorar a precisão dos testes e automatizar substancialmente os processos de avaliação de sistemas ferroviários. O SATS é projetado para lidar com a complexidade crescente dos sistemas baseados em microprocessadores, que tornaram os métodos tradicionais de teste manual insuficientes.

As motivações para o uso do *software* são diversas e incluem a necessidade de aumentar a precisão dos testes e reduzir o tempo de campo. O SATS permite que os engenheiros realizem testes completos de diferentes configurações de campo, proporcionando uma visão detalhada do sistema de sinalização como mostra a Figura 1.

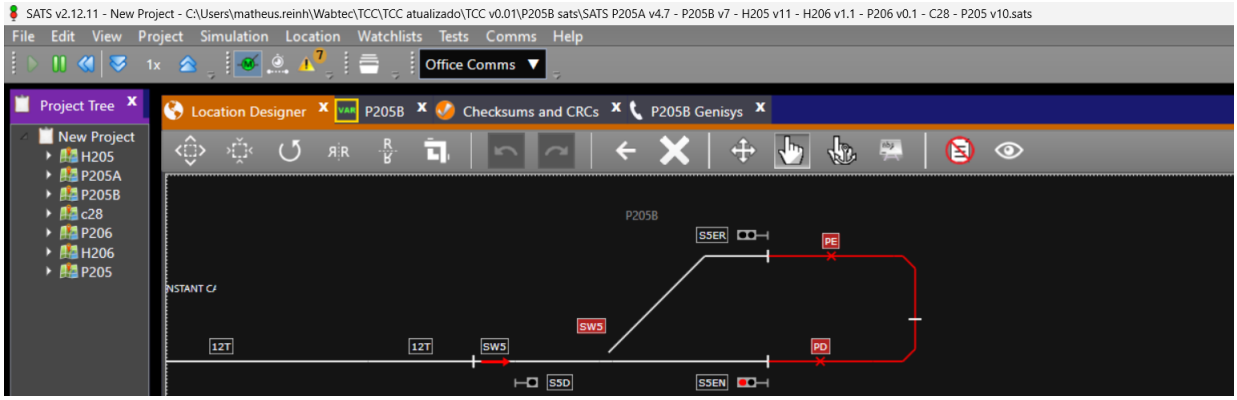

Figura 1 – Aba *Location Designer* do SATS

Fonte: Autor

A flexibilidade do programa em suportar múltiplos equipamentos e protocolos de comunicação, como ATCS e Genisys, torna-o uma ferramenta versátil para diversas aplicações ferroviárias. Assim, permite a comunicação entre Painéis de Controle Local e o *software* aplicativo do ELIX simulado na aplicação, como exemplifica a Figura 2 o que facilita o desenvolvimento da aplicação deste trabalho, sendo possível realizá-lo completamente em ambiente virtual.

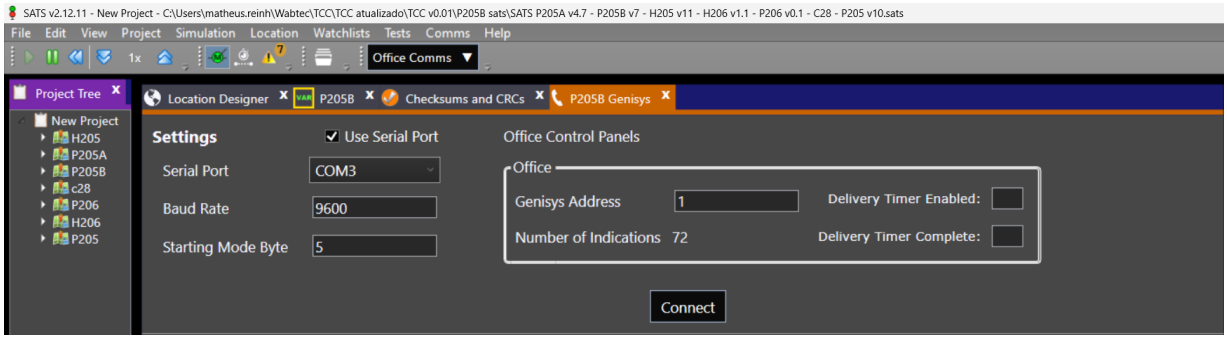

Figura 2 – Comms SATS

Fonte: Autor

Além disso, o suporte rápido e especializado oferecido pela equipe da Princeton Consultants garante que os usuários possam resolver dúvidas e problemas de forma eficiente.

3 Desenvolvimento

Este capítulo apresenta o desenvolvimento do projeto, que envolve a criação do *driver* para o protocolo Genisys e da Interface Humano-Máquina, que juntos formam o Painel de Controle Local. A Figura 3 é um diagrama que representa, de maneira geral, o sistema em que a aplicação estará inserida e irá interagir.

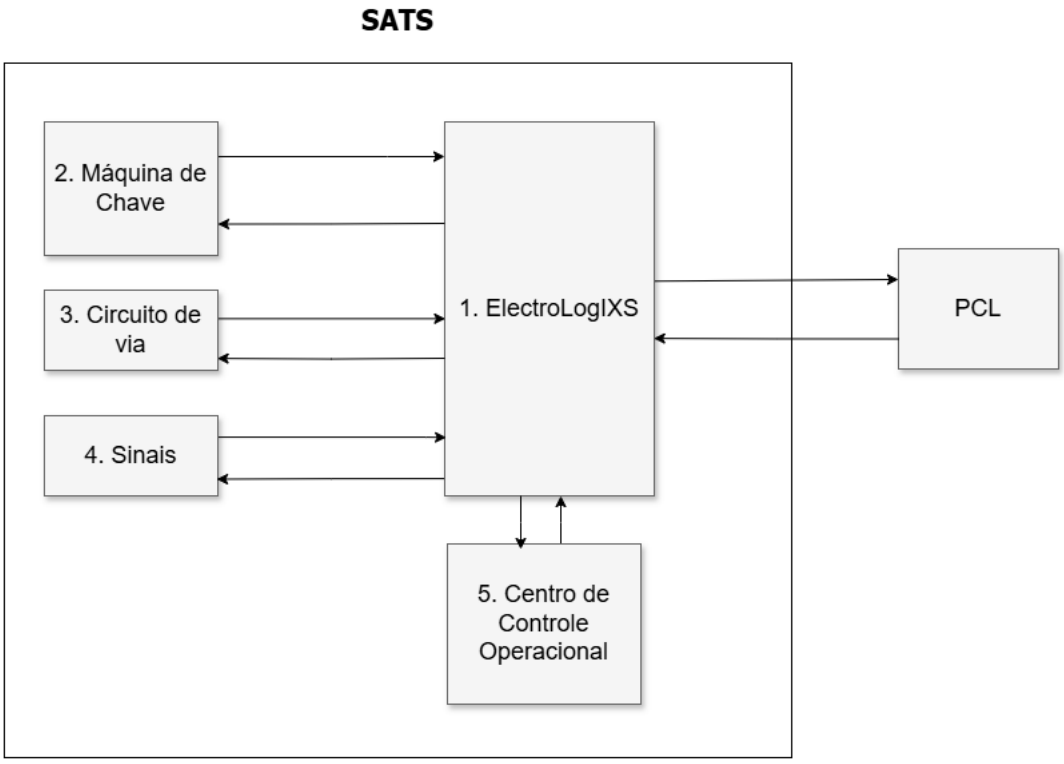

Figura 3 – Sistema Completo

Fonte: Autor

- 1. ElectroLogIXS: O ElectroLogIXS é um sistema avançado de controle e sinalização ferroviária desenvolvido pela General Electric. Ele é projetado para gerenciar e monitorar operações ferroviárias, garantindo segurança e eficiência. O sistema suporta uma ampla gama de aplicações, incluindo controle de passagens em nível, detecção de movimento, monitoramento de entradas vitais e controle de saída de relés vitais. Além disso, o ElectroLogIXS incorpora comunicação serial vital, que permite a integração com outros sistemas de de controle, inclusive o Painel de Controle Local desenvolvido neste trabalho;

- 2. Máquina de Chave: A máquina de chave é um dispositivo eletromecânico utilizado para mover e fixar os trilhos em uma posição específica, permitindo que os trens mudem de uma linha para outra. Essas máquinas são essenciais para a operação segura e eficiente das ferrovias, pois garantem que os trilhos estejam corretamente

alinhados antes da passagem de um trem. As máquinas de chave podem ser operadas manualmente ou automaticamente, e são frequentemente integradas a sistemas de controle centralizados para monitoramento e operação remota;

- 3. Circuito de Via: O circuito de via é um sistema elétrico utilizado para detectar a presença de trens em um determinado trecho da via férrea. Ele funciona criando uma diferença de potencial entre os trilhos, que é interrompida quando as rodas metálicas do trem entram em contato com os trilhos, criando um curto-circuito. Esse curto-circuito é detectado por um relé, que informa ao sistema de controle que o trecho está ocupado. Os circuitos de via são fundamentais para a sinalização ferroviária, pois ajudam a evitar colisões e garantem que os trens mantenham uma distância segura uns dos outros;

- 4. Sinais: Os sinais ferroviários são dispositivos visuais utilizados para comunicar informações críticas aos maquinistas, como permissões de avanço, limites de velocidade e avisos de perigo. Existem vários tipos de sinais, incluindo sinais luminosos, sinais de velocidade, sinais de manobra e sinais de advertência. Cada sinal tem um significado específico e é projetado para garantir a segurança e a eficiência das operações ferroviárias. Os sinais são posicionados ao longo da via férrea e nas proximidades das estações para orientar os maquinistas em suas rotas;

- 5. Centro de Controle Operacional: O Centro de Controle Operacional (CCO) é a central responsável pelo monitoramento e gerenciamento das operações ferroviárias em tempo real. Ele é equipado com sistemas de videomonitoramento, comunicação e análise de dados, permitindo que os operadores monitorem o tráfego ferroviário, respondam a incidentes e tomem decisões informadas para garantir a segurança e a eficiência das operações. Através do CCO é possível alinhar rotas, mover máquinas de chave, interditar e desinterditar circuitos, sendo assim, ele é essencial para a coordenação de atividades, redução de riscos e otimização de recursos, desempenhando um papel vital na gestão das operações ferroviárias.

Todos os componentes listados acima estarão simulados na aplicação SATS, que consegue se comunicar via protocolo Genisys. Através do programa Virtual Serial Ports, é possível estabelecer uma comunicação serial entre duas portas do computador, as quais serão acessadas, uma pelo simulador e outra pela aplicação desenvolvida.

3.1 Fluxograma do Código

Esta seção irá abordar o Programa desenvolvido, explicitando as duas frentes de desenvolvimento da aplicação, tanto a criação do *driver* para comunicação Genisys quanto da Interface Humano-Máquina. O código foi modulado em sete arquivos, sendo

cinco em Python e dois em linguagem Kivy. Os módulos são, main.py, genisys_driver.py, functions.py, supervisory.py, connection.py, supervisory.kv e connection.kv. A Figura 4 mostra o fluxograma principal do código que será tratado no decorrer do capítulo.

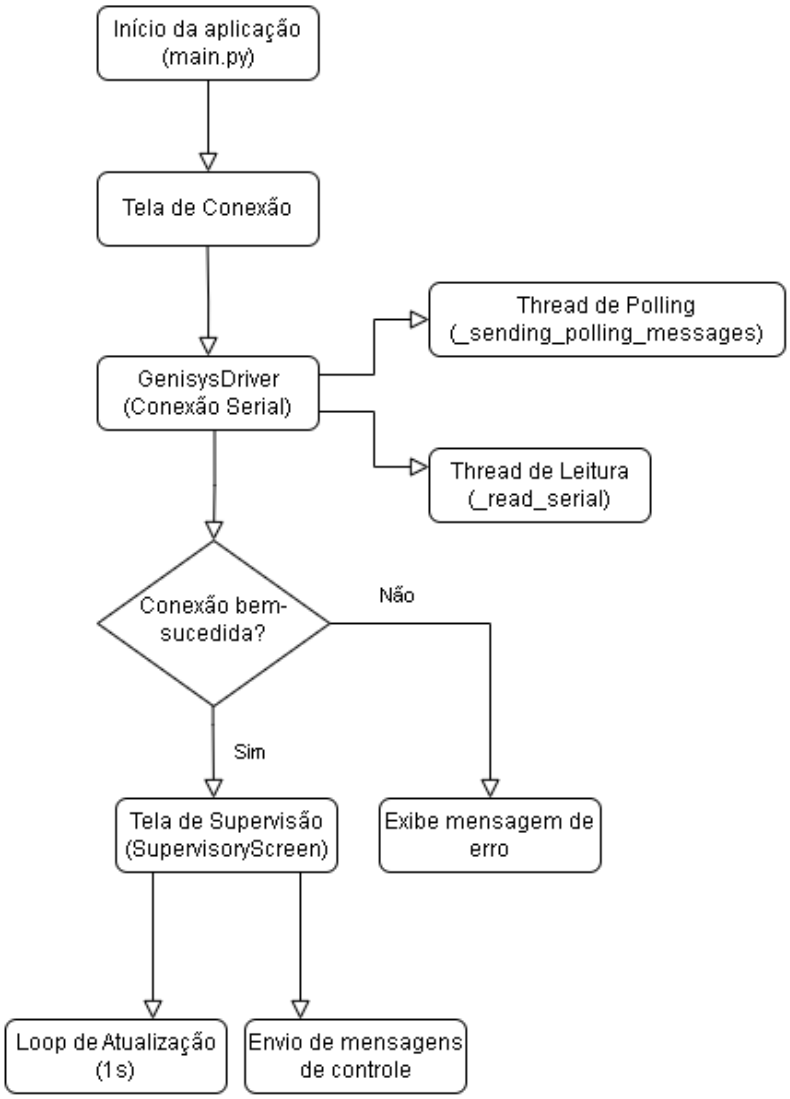

Figura 4 – Fluxograma do código

Fonte: Autor

3.1.1 Inicialização e Conexão

O sistema inicia com a execução do aplicativo GenisysApp, que carrega a tela de conexão, ConnectionScreen. Nesta etapa, o usuário seleciona a porta serial e o *baudrate*, e ao confirmar, o *driver*, GenisysDriver, é instanciado para estabelecer a comunicação. Se a conexão é bem-sucedida, o sistema redireciona para a tela de supervisão SupervisoryScreen,

onde o *driver* inicia duas *threads*: uma para leitura contínua de dados da serial e outra para envio periódico de comandos de *Poll* e *Recall*, mantendo a comunicação ativa.

3.1.2 Processamento de Dados e Atualização de Estado

A *thread* de leitura serial captura mensagens completas, sempre terminadas por 0xF6, e as encaminha para processamento. Mensagens do tipo Indicação, 0xF2, são tratadas para extrair o estado das variáveis: o conteúdo é, convertido em uma *string* global de zeros e uns e armazenado com sincronização via *lock* para evitar conflitos entre *threads*. Paralelamente, a *thread* de *Polling* envia solicitações periódicas para atualizar o estado do sistema. Após cada atualização de indicação, um *Acknowledge and Poll*, é enviado para confirmar o recebimento da mensagem.

3.1.3 Interface de Supervisão e Interação

Na tela de supervisão, um temporizador do Kivy (Clock.schedule_interval) atualiza a interface a cada segundo. Cada componente (LEDs, MCHs, botões) é mapeado para um *bit* específico do *status* (variável global), usando a função cal_indication_index para corrigir índices. Botões de controle, ao serem pressionados, acionam o método send_control_message, que envia comandos serializados via *driver*. O sistema garante confiabilidade com técnicas como CRC, escaping de *bytes* e sincronização de *threads*, assegurando que a interface reflita fielmente o estado físico dos dispositivos em tempo real.

3.2 Desenvolvimento do *Driver* Genisys

O protocolo Genisys, desenvolvido pela Union Switch & Signal, é amplamente utilizado para comunicação entre dispositivos de campo (como controladores lógicos programáveis) e sistemas de supervisão em infraestruturas ferroviárias. Projetado originalmente para comunicação serial, possui adaptação para TCP/IP, mantendo características robustas como *escaping* de dados e verificação de integridade via CRC-16.

Para a realização do processo de *sniffing* foi necessário escolher algum projeto específico, para facilitar a compreensão das mensagens trocadas entre o ELIX e o Painel de Controle Local. A locação escolhida foi a de uma Pera de Carregamento[1] de trens presente em Barão de Cocais, que em sua configuração possui todos os elementos necessários para desenvolver um *driver* que seja replicável para qualquer cenário. Sendo assim, com o *software* aplicativo do ELIX, referente a Pera de Carregamento, foi possível definir um mapa de *bits*, um Painel de Controle Local no sistema Legado, e um arquivo SATS configurado para os testes.

[1] Local de carregamento de vagões ferroviários.

O simulador SATS é carregado com o mesmo *software* aplicativo que é instalado no CLP de campo, além de ser necessário desenhar toda a configuração que será simulada para os testes do projeto. Tendo toda a simulação estabelecida, a comunicação entre o SATS e o PCL gera alguns arquivos de *log* para a análise das mensagens trocadas. O arquivo consiste na mensagem enviada, seguido do mapa de *bits* indicando qual variável está no estado *on* e qual está no estado *off*, como ilustra a Figura 5 abaixo.

Figura 5 – *Log* do SATS

Fonte: Autor

Inicialmente, essas informações foram utilizadas para entender como se dava o endereçamento das indicações dentro do pacote de mensagens enviados do CLP para o PCL. Seguindo o processo de alterar o *status* de alguma variável conhecida do mapa de *bits* e observar qual a alteração no pacote de mensagem, como se observa na Figura 6.

```
Sent Indications
Raw Data: F2-01-00-01-01-00-02-21-03-00-04-10-05-D8-06-1E-07-C0-08-3A-E0-05-D8-00-F6
        LPLOCALKE
ON      LPCENTRALKE


Sent Indications
Raw Data: F2-01-00-01-01-00-02-21-03-00-04-10-05-D8-06-1D-07-C0-08-3A-E0-05-EB-00-F6
ON      LPLOCALKE
        LPCENTRALKE
```

Figura 6 – Mudança de *Status*

Fonte: Autor

O *status* a ser observado estava relacionado ao estabelecimento do modo de con-

trole local, portanto foi possível observar que a mensagem sofreu alteração somente em alguns octetos, sendo este o octeto que segue após o valor 0x06 em ambas as mensagens. Observando o mapa de *bits* e dividindo-o em octetos. As indicações de modo de controle local e modo de controle central, são as de endereço 49 e 50, respectivamente, como mostra a Figura 7, portanto estão endereçadas exatamente dentro do sexto octeto.

Indication 49	LPLOCALKE
Indication 50	LPCENTRALKE
Indication 51	LPSYSHELTHKE
Indication 52	LP110VPOKE
Indication 53	LP12VDCPOKE
Indication 54	LPS5DFRKE
Indication 55	LPS5DRKE
Indication 56	LPS5ENRKE

Figura 7 – Sexto Octeto do Mapa de *Bits*

Fonte: Autor

Seguindo essa fórmula para inúmeros casos de indicações conhecidas do mapa de *bits*, tornou-se possível interpretar o conteúdo das mensagens de indicação recebidas, passando a ser possível criar um supervisório que monitora as indicações enviadas pelo CLP. Entretanto, como o objetivo principal do trabalho é criar uma Interface Humano-Máquina que consiga tanto receber informações das indicações quanto enviar mensagens de controle, mostrou-se necessário aprofundar o estudo e a compreensão do pacote de *bytes* do protocolo. Portanto, a identificação das demais partes da mensagem, como o *header*, o terminador e o processo de CRC se tornaram essenciais para que criar mensagens aceitas pelo CLP.

Para a interpretação do processo de CRC de o trabalho desenvolvido em [2], serviu como base para compreensão do protocolo. Cruzando as informações desse artigo com os *logs* obtidos pelo SATS e o monitor Serial da aplicação legado, foi possível entender a criação do CRC e também o significado dos demais octetos das mensagens do protocolo e definir a estrutura do *frame* Genisys.

3.2.1 Estrutura do *Frame* Genisys

Nesta seção a estrutura do *frame* Genisys será detalhada e explicada octeto por octeto. As mensagens do protocolo seguem um formato específico, observado na Figura 8:

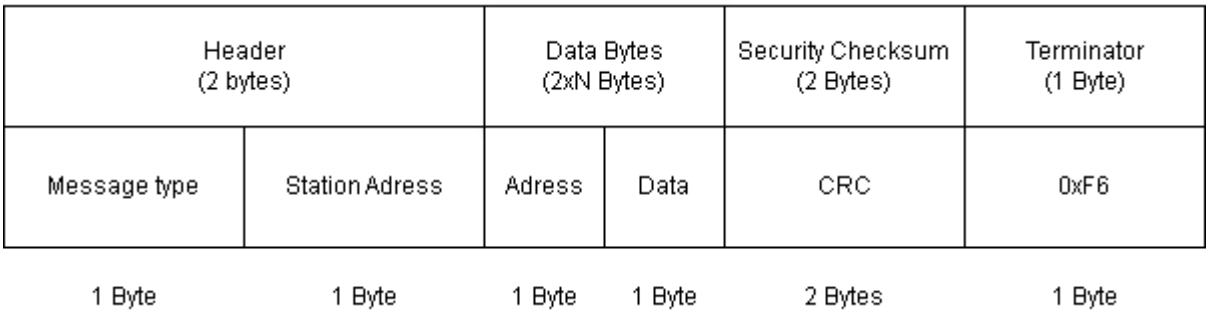

Figura 8 – Estrutura do *frame* Genisys

Fonte: Autor

A Figura 9 apresenta uma mensagem recebida no monitor serial. Por se tratar de uma mensagem de indicação, possui o *frame* completo, com *header*, *payload* com tamanho correspondendo ao número de octetos de indicação do *software*, que são 8 para esse caso, o CRC e o terminador da mensagem.

Figura 9 – Mensagem Bruta no Monitor Serial

Fonte: Autor

- Identificador de Mensagem: Um octeto no intervalo de 0xF1 a 0xFE, que inicia a comunicação e identifica o tipo de mensagem.
- Endereço do Escravo: Um único *byte* que identifica o dispositivo destinatário.
- *Payload*: Composto por dados de comprimento variável, organizados em pares de *bytes* de 8 *bit* seguindo o padrão *endereço/dado*.
- CRC-16: Dois *bytes* para garantir o conteúdo do pacote que aparecem em algumas mensagens.
- Terminador: Octeto fixo 0xF6, indicando o fim da mensagem.

O protocolo conta com o processo de *escaping* de octetos com valor superior a 0xF0 (excluindo o cabeçalho e o terminador). O processo substitui o octeto original por dois octetos: 0xF0 seguido do *nibble* inferior do valor original. Por exemplo:

$$\texttt{0xF4} \rightarrow \texttt{0xF0 0x04}$$

Em mensagens que incluem CRC, este é calculado utilizando o polinômio

$$x^{16} + x^{15} + x^2 + 1$$

transmitido em ordem *little-endian*. O cálculo abrange todos os octetos da mensagem, incluindo o cabeçalho, mas excluindo o terminador. Para otimizar o processo, utiliza-se uma tabela de *lookup* pré-computada, que permite calcular o CRC em tempo constante por *byte*. A tabela contém 256 entradas, uma para cada possível valor de 8 *bits*, na qual cada entrada é o CRC resultante do polinômio aplicado ao *byte* correspondente. É importante destacar que o CRC é calculado antes da aplicação do mecanismo de escape e os *bytes* do CRC também devem ser escapados, se necessário.

O *payload*, quando presente, consiste em uma sequência de pares endereço/dado, onde cada par contém um *byte* de endereço seguido de um *byte* de dado. Ambos os componentes devem obedecer às regras de escape.

3.3 Aplicação do *Driver* Genisys no Código

O desenvolvimento do sistema de supervisão e controle baseado no protocolo Genisys envolveu a implementação de um *driver* especializado, responsável por gerenciar a comunicação serial com os dispositivos de campo. Este *driver* foi projetado seguindo princípios de modularidade e eficiência, garantindo uma integração fluida com a interface gráfica desenvolvida em Kivy.

3.3.1 Inicialização e Ciclo de Vida do *Driver*

A inicialização do *driver* Genisys foi estruturada de forma a otimizar o uso de recursos e garantir a robustez do sistema. A instanciação do objeto *driver* ocorre de maneira condicional, somente após a confirmação da conexão serial. Essa abordagem, conhecida como inicialização tardia, previne o acesso a métodos não inicializados e permite uma configuração dinâmica dos parâmetros de comunicação.

A estrutura principal da aplicação, definida na classe GenisysApp, demonstrada no Algoritmo 3.1, utiliza o gerenciador de telas ScreenManager para alternar entre a tela de conexão, ConnectionScreen, e a tela de supervisão SupervisoryScreen. O *driver* é inicializado como *None* e só é instanciado quando o usuário confirma os parâmetros de conexão.

```
1 class GenisysApp(App):
2     def build(self):
3         self.driver = None  # Inicializa  o tardia
4         sm = ScreenManager()
5         sm.add_widget(ConnectionScreen(name='connection'))
6         sm.add_widget(SupervisoryScreen(name='supervisory'))
7         return sm
```

Algoritmo 3.1 – Estrutura principal da aplicação

3.3.2 Estabelecimento de Conexão

A instanciação do *driver* ocorre na tela de conexão, onde o usuário define a porta serial e o *baudrate*. No Algoritmo 3.2, observa-se o método *connect*, que é responsável por criar o objeto *driver* e iniciar a comunicação serial. Durante esse processo, são realizadas validações dos parâmetros de conexão, garantindo que a comunicação seja estabelecida de forma segura e confiável.

```
def connect(self, port: str, baudrate: str):
    try:
        app = App.get_running_app()
        app.driver = GenisysDriver(  # Cria  o condicional
            port=port,
            baudrate=int(baudrate),
            max_control_value=128
        )
        app.driver.connect()  # Handshake serial
        self.manager.current = 'supervisory'
        supervisory = self.manager.get_screen('supervisory')
        supervisory.initialize(app.driver)  # Inje  o de depend ncia
```

Algoritmo 3.2 – Criação do driver na conexão

Após a conexão ser estabelecida, o sistema transiciona para a tela de supervisão, onde o *driver* é passado como dependência para a inicialização dos componentes gráficos. Esse mecanismo de injeção de dependência garante que a interface gráfica tenha acesso direto ao *driver*, permitindo a atualização dinâmica dos estados dos dispositivos.

3.3.3 *Thread* de Comunicação Serial

Para garantir a eficiência e a responsividade do sistema, o *driver* Genisys emprega duas *threads* principais: uma para leitura contínua de dados da porta serial e outra para o envio periódico de mensagens de *polling*. A *thread* de leitura é responsável por capturar as mensagens enviadas pelos dispositivos e processá-las, enquanto a *thread* de *polling* mantém a comunicação ativa, enviando mensagens de verificação e solicitação de estado. O Algoritmo 3.3 apresenta a inicialização das threads.

```
def connect(self) -> None:
    self._running = True
    # Thread de leitura cont nua
    self._read_thread = threading.Thread(target=self._read_serial,
        daemon=True)
    # Thread de polling peri dico
    self._poll_thread = threading.Thread(target=self.
        _send_polling_messages, daemon=True)
```

```
7       self._read_thread.start()
8       self._poll_thread.start()
```

Algoritmo 3.3 – Gerenciamento de threads de comunicação

A utilização de *threads* permite que o sistema opere de forma assíncrona, garantindo que a interface gráfica permaneça responsiva mesmo durante operações de comunicação intensivas.

3.3.3.1 Política de *Polling*

A estratégia de *polling* adotada pelo *driver* combina mensagens regulares de verificação, *poll* com solicitações periódicas de estado completo *recall*. O intervalo entre as mensagens de *polling* é definido por uma sequência cíclica (`POLL_INTERVALS`), como observa-se no Algoritmo 3.4, baseada na sequência observada em *logs* no monitor serial do sistema legado.

```
1  POLL_INTERVALS = [4, 4, 3]  # Sequ ncia c clica de intervalos
2
3  def _send_polling_messages(self) -> None:
4      while self._running and self.connected:
5          with self._lock:  # Sincroniza  o de thread
6              self._safe_write(self.POLL_MESSAGE)
7              self._poll_count += 1
8
9              if self._poll_count >= self.POLL_INTERVALS[self.
                   _current_interval_index]:
10                 self._safe_write(self.RECALL_MESSAGE)  # Request full
                       state
11                 self._poll_count = 0
12                 self._current_interval_index = (self.
                       _current_interval_index + 1) % 3
13         time.sleep(1)  # Controle temporal
```

Algoritmo 3.4 – Lógica de polling adaptativo

Essa abordagem garante que o sistema mantenha uma comunicação ativa com os dispositivos, verificando periodicamente seu estado e solicitando atualizações completas quando necessário.

3.3.4 Processamento de Indicações

As mensagens de indicação são o mecanismo pelo qual os dispositivos informam seu estado atual ao sistema. Quando uma mensagem de indicação é recebida, o *driver* realiza uma série de operações para processar e atualizar o estado global do sistema, como observa-se no Algoritmo 3.5.

```
def _handle_indication(self, message: bytes) -> None:
    unescaped = reverse_escaping(message)  # Remove bytes de escape
    binary_content = self._parse_indication_content(unescaped)

    with self._lock:  # Critical section
        self.status = binary_content  # Estado global atualizado

    self._safe_write(self.ACK_POLL_MESSAGE)  # Confirma  o
    logger.info("Estado atualizado: %s", binary_content)
```

Algoritmo 3.5 – Tratamento de mensagens de indicação

O processamento de uma mensagem de indicação envolve a remoção dos *bytes* de escape, método reverse_escaping, a conversão do conteúdo da mensagem para uma representação binária e a atualização segura do estado global do sistema. Após o processamento, o *driver* envia uma mensagem de confirmação para garantir que a comunicação permaneça ativa.

3.3.5 Sincronização com a Interface

O Algoritmo 3.6 apresenta a inicialização da interface gráfica do sistema, que é atualizada periodicamente com base no estado atual dos dispositivos. Essa atualização é gerenciada por um *callback* temporal, que é acionado a cada segundo.

```
class SupervisoryScreen(Screen):
    def initialize(self, driver):
        self.driver = driver
        Clock.schedule_interval(self.update_all_images, 1)  # 1Hz

    def update_all_images(self, dt):
        status = self.driver.get_status()  # Obt m snapshot seguro
        self.update_all_led_images(status)
        self.update_all_mch_images(status)
        self.update_all_track_images(status)
```

Algoritmo 3.6 – Atualização periódica da interface

A atualização da interface é realizada de forma seletiva, onde apenas os componentes cujo estado foi alterado são redesenhados. Essa otimização reduz o consumo de recursos e melhora a responsividade do sistema.

3.4 Desenvolvimento da Interface Humano-Máquina

A Interface Humano-Máquina foi criada através do *framework* Kivy. Além da familiaridade do autor com o *framework*, a possibilidade de criar interfaces com linguagem

escrita, ao invés do modelo de objetos utilizado pelo Adobe Flash, o que facilita o uso de IA para geração de código, e também a possibilidade de fechar a solução e comercializá-la futuramente sem custo adicional, motivaram a escolha do *framework*.

O painel desenvolvido apresenta duas telas. A primeira, tem o objetivo de configurar alguns parâmetros como a porta COM que será utilizada e o *baudrate*. Além dessa configuração ela conta com um botão para inicializar a conexão. Caso a conexão seja estabelecida, a segunda tela é carregada, essa por sua vez, apresenta o supervisório da locação definida para o trabalho.

A atualização da tela do supervisório, possui uma lógica simples, pois é baseada na substituição de imagens de acordo com o *status* de suas respectivas variáveis. O Algortimo 3.7 apresenta o método que atualiza todas as imagens a cada um segundo, para manter o supervisório sempre atualizado.

```
1  class SupervisoryScreen(Screen):
2      connected = BooleanProperty(False)
3      Window.fullscreen = 'auto'
4
5      def initialize(self, driver):
6          """Inicializa a tela com a inst ncia do driver"""
7          self.driver = driver
8          self.connected = driver.connected
9          self.last_status = None
10         self.button_states = {}
11         # Inicia a atualiza  o peri dica da imagem
12         Clock.schedule_interval(self.update_all_images, 1)  # Atualiza a
               cada 1 segundo
13
14     def update_all_images(self, dt):
15         """Atualiza todas as imagens"""
16         new_status = self.driver.get_status()
17
18         if new_status is None:
19             return
20
21         if self.last_status is None or new_status != self.last_status:
22             self.last_status = new_status
23
24         self.update_all_led_images(new_status)  # Atualiza todas as
              imagens de LED
25         self.update_all_mch_images(new_status)  # Atualiza todas as
              imagens de MCH
26         self.update_all_track_images(new_status)  # Atualiza todas as
              imagens de Track
27         self.update_all_signal_images(new_status)  # Atualiza todas as
```

```
            imagens de Signal
28          self.update_all_button_backgrounds(new_status)  # Atualiza os
            planos de fundo dos bot es
```

Algoritmo 3.7 – Inizialização do Supervisório

Para tratar cada tipo de imagem do supervisório, foram criados métodos em Python e modelos de *widgets* em linguagem *Kivy* específicos de cada caso, os quais cada imagem é associada ao um endereço do mapa de *bits*.

3.4.1 LED's

Os LED's são utilizados em diversos contextos, podendo ser alarmes em geral, saúde de conexões ou qualquer indicação que dependa de apenas um *bit* para trazer uma informação. A Figura 10 e o Algoritmo 3.8, ilustram um exemplo.

Figura 10 – Exemplo de LED

Fonte: Autor

```
1           BoxLayout:
2               orientation: 'vertical'
3               size_hint: (0.1, 0.1)
4               pos_hint: {'x': 0.595, 'y': 0.02}
5               Image:
6                   id: LED_falha_MCH
7                   is_led: True
8                   size_hint: (1, 0.8)
9                   source: 'images/leds/LEDapagado.png'
10                  index: 19
11                  image_1: 'images/leds/rLED.png'
12                  image_0: 'images/leds/LEDapagado.png'
13              Label:
14                  text: '[b]FALHA[/b]'
15                  markup: True
16                  size_hint: (1, 0.2)
17                  font_size: 14
18                  color: (0, 0, 0, 1)
```

Algoritmo 3.8 – Definição de um LED em .kv

Todos foram definidos com um identificador "is_led", para que o método de atualização específico, apresentado no Algoritmo 3.9, consiga identificá-los quando percorre todo o arquivo .kv no momento em que é executado. Através do *widget BoxLayout* sua aplicação está sempre associada a um *Label* para descrever sua função. Possui apenas imagens de apagado e aceso, variando em 3 cores: verde, vermelho e amarelo.

```
1     def update_all_led_images(self, status):
2         """Atualiza todas as imagens de LED"""
3         for widget in self.walk():
4             if isinstance(widget, Widget) and hasattr(widget, 'is_led')
                  and widget.is_led:
5                 index = cal_indication_index(widget.index, 16)
6                 if len(status) > index:
7                     if status[index] == '1':
8                         widget.source = widget.image_1
9                     else:
10                        widget.source = widget.image_0
```

Algoritmo 3.9 – Método para atualização dos LED's

3.4.2 Sinais

Os sinais, ilustrados na Figura 11 e no Algoritmo 3.10 seguem uma lógica parecida com a dos LED's, contudo em determinados contextos da sinalização ferroviária mais de um aspecto do sinaleiro é ativado ao mesmo tempo, por exemplo, os chamados "vermelho sobre amarelo", estado em que ambos os Led's do sinal estão acesos. Diante dessa condição, é necessário uma lógica que observe casos como esse.

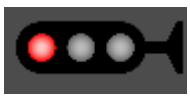

Figura 11 – Exemplo de Sinal

Fonte: Autor

```
1         Image:
2             id: signal_S5D
3             is_signal: True
4             size_hint: (0.12, 0.12)
5             pos_hint: {'x': 0.393, 'y': 0.375}
6             source: 'images/signal/apagado.png'
7             index_red: 55
8             index_flashred: 54
9             index_yellow: 0
```

```
            index_flashyellow: 0
            index_green: 0
            image_red: 'images/signal/red.png'
            image_flashred: 'images/signal/red.png'
            image_yellow: 'images/signal/yellow.png'
            image_flashyellow: 'images/signal/yellow.png'
            image_green: 'images/signal/green.png'
            image_redyellow: 'images/signal/redyellow.png'
            image_apagado: 'images/signal/apagado.png'
            canvas.before:
                PushMatrix
                Rotate:
                    angle: -90
                    origin: self.center
            canvas.after:
                PopMatrix
```

Algoritmo 3.10 – Definição de um sinal em .kv

Portanto, observa-se que cada aspecto do sinal possui a própria imagem associada, e também um endereço associado no mapa de *bits*.

A seguir, o Algoritmos 3.11 e o Algoritmo 3.12 apresentam como acontece a atualização das imagens dos sinais.

```
    def update_all_signal_images(self, status):
        """Atualiza todas as imagens de Signal"""
        for widget in self.walk():
            if isinstance(widget, Widget) and hasattr(widget, 'is_signal
                ') and widget.is_signal:
                #logger.debug(f"\033[91mAtualizando track: {widget
                    }\033[0m")
                image_source = self.update_signal(widget, status)
                if image_source:  # Verifica se image_source n o
                    None
                    widget.source = image_source
                    #widget.canvas.ask_update()  # For a a
                        atualiza  o do widget
```

Algoritmo 3.11 – Método para atualização dos Sinais

O primeiro método inicializa a atualização de todos *widgets*, que possuem o identificador "is _signal", que por sua vez irá iniciar o segundo método que trata individualmente cada sinal.

```
    def update_signal(self, widget, status):
        """L gica espec fica para signal"""
```

```
3           if widget.index_red != 0:
4               index_red = cal_indication_index(widget.index_red, 16)
5               if len(status) <= index_red:
6                   logger.error("\033[91mStatus string length is
                        insufficient for the required indices\033[0m")
7                   return widget.source  # Retorna a imagem atual se o
                        status n o for v lido
8               if status[index_red] == '1':
9                   return widget.image_red
10
11          if widget.index_flashred != 0:
12              index_flashred = cal_indication_index(widget.index_flashred,
                    16)
13              if len(status) <= index_flashred:
14                  logger.error("\033[91mStatus string length is
                        insufficient for the required indices\033[0m")
15                  return widget.source  # Retorna a imagem atual se o
                        status n o for v lido
16              if status[index_flashred] == '1':
17                  return widget.image_flashred
18
19          if widget.index_yellow != 0:
20              index_yellow = cal_indication_index(widget.index_yellow, 16)
21              if len(status) <= index_yellow:
22                  logger.error("\033[91mStatus string length is
                        insufficient for the required indices\033[0m")
23                  return widget.source  # Retorna a imagem atual se o
                        status n o for v lido
24              if status[index_yellow] == '1':
25                  return widget.image_yellow
26
27          if widget.index_flashyellow != 0:
28              index_flashyellow = cal_indication_index(widget.
                    index_flashyellow, 16)
29              if len(status) <= index_flashyellow:
30                  logger.error("\033[91mStatus string length is
                        insufficient for the required indices\033[0m")
31                  return widget.source  # Retorna a imagem atual se o
                        status n o for v lido
32              if status[index_flashyellow] == '1':
33                  return widget.image_flashyellow
34
35          if widget.index_green != 0:
36              index_green = cal_indication_index(widget.index_green, 16)
37              if len(status) <= index_green:
38                  logger.error("\033[91mStatus string length is
                        insufficient for the required indices\033[0m")
```

```
39                  return widget.source  # Retorna a imagem atual se o
                        status n o for v lido
40              if status[index_green] == '1':
41                  return widget.image_green
42
43          else:
44              return widget.image_apagado
```

Algoritmo 3.12 – Método com a lógica de atualização dos Sinais

O segundo método verifica se cada endereço do mapa de *bits* foi setado para algum valor diferente de 0. Essa condição foi aplicada para que em futuras edições da interface para diferentes configurações de pátio, o código se mantenha escalável, não necessitando de apagar indicadores de aspectos de sinal que não serão utilizados, podendo apenas configura-los com o valor 0 no respectivo arquivo .kv. Atendendo a condição de possuir endereço maior do que zero, o sinal será alterado caso o *status* correspondente esteja em nível lógico 1.

3.4.3 Circuito de Via

Os blocos de circuito de via são os que representam a linha que o trem percorre de fato. Basicamente devem monitorar se o circuito está ocupado ou livre. A ocupação é sempre representada pelo bloco do circuito em vermelho, como na Figura 12. Ademais, também são utilizados para visualização do conceito de rotas, que podem estar alinhadas ou apenas requisitadas. Portanto todos os casos possuem sua própria imagem associada, como apresenta o Algoritmo 3.13.

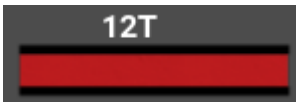

Figura 12 – Exemplo de Circuito de Via Ocupado

Fonte: Autor

```
1           Image:
2               id: track1
3               is_track: True
4               pos_hint: {'x': -0.13, 'y': -0.02}
5               source: 'images/tracks/trackpreta.png'
6               occupation_index: 39
7               requested_route_index: 30
8               aligned_route_index: 26
9               image_occupied: 'images/tracks/trackvermelha.png'
10              image_aligned: 'images/tracks/trackverde.png'
```

```
11              image_requested: 'images/tracks/trackamarela.png'
12              image_track: 'images/tracks/trackpreta.png'
```

Algoritmo 3.13 – Definição de um Circuito de Via em .kv

Como as imagens irão monitorar ocupação do circuito de via, requisição e alinhamento de rota, são necessários três diferentes endereços do mapa de *bits*, um para cada variável a ser observada, além das imagens associadas a cada um desses.

A seguir, o Algoritmo 3.14, apresenta a lógica para atualização de imagens de circuito de via.

```
1     def update_all_track_images(self, status):
2         """Atualiza todas as imagens de MCH"""
3         for widget in self.walk():
4             if isinstance(widget, Widget) and hasattr(widget, 'is_track'
                  ) and widget.is_track:
5                 #logger.debug(f"\033[91mAtualizando track: {widget
                      }\033[0m")
6                 image_source = self.update_track(widget, status)
7                 if image_source:  # Verifica se image_source n o
                      None
8                     widget.source = image_source
9                     #widget.canvas.ask_update()  # For a a
                          atualiza  o do widget
10
11    def update_track(self, widget, status):
12        """L gica espec fica para track"""
13        occupation_index = cal_indication_index(widget.occupation_index,
              16)
14        requested_route_index = cal_indication_index(widget.
              requested_route_index, 16)
15        aligned_route_index = cal_indication_index(widget.
              aligned_route_index, 16)
16
17        if len(status) <= max(occupation_index, requested_route_index,
              aligned_route_index):
18            logger.error("\033[91mStatus string length is insufficient
                  for the required indices\033[0m")
19            return widget.source  # Retorna a imagem atual se o status
                  n o for v lido
20
21        if status[occupation_index] == '1':
22            return widget.image_occupied
23        elif status[occupation_index] == '0':
24            if status[aligned_route_index] == '1':
25                return widget.image_aligned
```

```
26              elif status[requested_route_index] == '1':
27                  return widget.image_requested
28              else:
29                  return widget.image_track
30          return widget.source
```

Algoritmo 3.14 – Método para atualização do Circuito de Via

A ocupação por ser uma informação crítica é a primeira condição a ser observada, e caso nenhum das demais condições sejam atendidas, é retornada a imagem da via desocupada e sem rota alinhada ou requisitada como na Figura 13.

Figura 13 – Exemplo de Circuito de Via Livre

Fonte: Autor

3.4.4 Máquina de Chave

A máquina de chave é um caso particular do bloco de circuito de via, pois essa possui a mesma lógica quanto à ocupação e o alinhamento e requisição de rotas. Entretanto, possui a peculiaridade de apresentar para qual sentido a ponta de agulha da máquina de chave está alinhada, podendo estar nas posições normal, reverso ou sem indicação. A Figura 14 representa uma máquina de chave em normal e com rota alinhada. O Algoritmo 3.15 apresenta todas as definições associadas a uma máquina de chave.

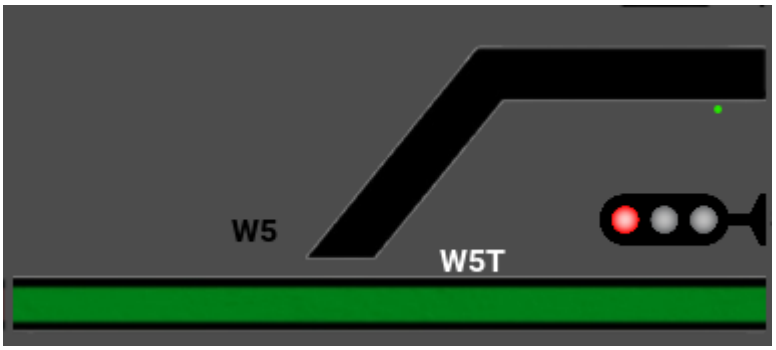

Figura 14 – Exemplo de Máquina de Chave

Fonte: Autor

```
1           Image:
```

```
            id: mch1
            is_mch: True
            size_hint: (0.45, 0.49)
            pos_hint: {'x': 0.335, 'y': 0.31}
            source: 'images/mch/mchsemindicacao.png'
            normal_index: 17
            reverse_index: 18
            occupation_index: 40
            requested_route_index: 30
            aligned_route_index: 26
            image_normaloccupied: 'images/mch/mchnormalvermelho.png'
            image_normalrequested: 'images/mch/mchnormalamarelo.png'
            image_normalaligned: 'images/mch/mchnormalverde.png'
            image_normaltimelocking: 'images/mch/mchnormalamarelo.png'
            image_normalmch: 'images/mch/mchnormal.png'
            image_reverseoccupied: 'images/mch/mchreversovermelha.png'
            image_reverserequested: 'images/mch/mchreversoamarelo.png'
            image_reversealigned: 'images/mch/mchreversoverde.png'
            image_reversemch: 'images/mch/mchreverso.png'
```

Algoritmo 3.15 – Definição da máquina de chave em .kv

Por ter muitas variações a serem apresentadas, as MCH's são as que possuem maior quantidade de endereços do mapa de *bits*, consequentemente maior quantidade de imagens associadas.

A seguir, o Algoritmo 3.16 e o Algoritmo 3.17 apresentam a lógica para atualização de imagens de máquina de chave.

```
    def update_all_mch_images(self, status):
        """Atualiza todas as imagens de MCH"""
        for widget in self.walk():
            if isinstance(widget, Widget) and hasattr(widget, 'is_mch')
                and widget.is_mch:
                #logger.debug(f"\033[91mAtualizando MCH: {widget}\033[0m
                    ")
                image_source = self.update_mch(widget, status)
                if image_source:  # Verifica se image_source n o
                    None
                    widget.source = image_source
                    #widget.canvas.ask_update()  # For a a
                        atualiza  o do widget
```

Algoritmo 3.16 – Método para atualização da Máquina de Chave

Assim como a lógica dos blocos de circuito de via, um primeiro método inicializa a atualização das imagens, verificando se possui o identificador "is _mch". Atendendo à

condição, o método de atualização de cada máquina de chave é chamado.

```
    def update_mch(self, widget, status):
        """L gica espec fica para MCH"""
        normal_index = cal_indication_index(widget.normal_index, 16)
        reverse_index = cal_indication_index(widget.reverse_index, 16)
        occupation_index = cal_indication_index(widget.occupation_index,
            16)
        requested_route_index = cal_indication_index(widget.
           requested_route_index, 16)
        aligned_route_index = cal_indication_index(widget.
           aligned_route_index, 16)

        #logger.debug(f"\033[91mIndices MCH: normal={normal_index},
           reverse={reverse_index}, occupation={occupation_index},
           requested_route={requested_route_index}, aligned_route={
           aligned_route_index}\033[0m")

        # Verifique se o status tem o comprimento necess rio
        if len(status) <= max(normal_index, reverse_index,
           occupation_index, requested_route_index, aligned_route_index)
           :
            logger.error("\033[91mStatus string length is insufficient
               for the required indices\033[0m")
            return widget.source  # Retorna a imagem atual se o status
               n o for v lido

        # Implementar a l gica com base nos valores dos  ndices
        if status[normal_index] == '1':
            if status[occupation_index] == '1':
                return widget.image_normaloccupied
            elif status[occupation_index] == '0':
                if status[aligned_route_index] == '1':
                    return widget.image_normalaligned
                elif status[requested_route_index] == '1':
                    return widget.image_normalrequested
                else:
                    return widget.image_normalmch
```

Algoritmo 3.17 – Método com a lógica para atualização de cada Máquina de Chave

Antes de verificar as condições das variáveis, existe uma conferência se a *string "status"* possui tamanho suficiente para acessar o endereço requerido, caso contrário retorna a imagem atual.

Atendendo aos requisitos de endereço válido, o método verifica em primeira instância se a máquina está em normal ou em reverso, para que as demais condições sejam aplicadas

de acordo com esse *status*. Não atendendo a nenhum dos dois, a imagem referente a falta de indicação do *status* da máquina de chave é carregada. Caso a condição de normal ou reverso seja satisfeita, a lógica passa a ser semelhante a de um bloco de circuito de via comum, observando ocupação, requisição e alinhamento de rota.

3.4.5 Botões

Os botões são utilizados para o envio de mensagens de controle, mas possuem também a propriedade de indicar o *status* de algum *bit* que lhe seja endereçado, atuando de maneira similar ao LED quanto a indicação. Eles estão associados a um *widget FloatLayout*, para poder posicionar também um *Label* que informa sua atribuição. Na Figura 15 e no Algoritmo 3.18, está presente a representação do botão que deve ser pressionado para enviar o comando de requisição de modo de controle local, e também a indicação de modo de controle local estabelecido por estar com o aspecto de aceso. Portanto, atuam tanto como indicadores quanto controladores.

Figura 15 – Exemplo de Botão

Fonte: Autor

```
1     FloatLayout:
2             id: Local
3             size_hint: (None, None)
4             size: (120, 120)
5             pos_hint: {'x': 0.40, 'y': 0.7}
6             Button:
7                 background_normal: 'images/buttons/g_off.png'  # Imagem
                      de fundo padr o
8                 background_down: 'images/buttons/g_off_down.png'  #
                      Imagem de fundo quando o bot o    pressionado
9                 padding: [0.01, 0.01, 0.05, 0.05]  # Ajuste os valores
                      conforme necess rio
10                is_button: True
11                size_hint: (None, None)  # Desabilita o ajuste
                      autom tico de tamanho
12                size: (130, 130)  # Define o tamanho do bot o (largura,
                       altura)
13                pos_hint: {'center_x': 0.5, 'center_y': 0.5}  #
                      Centraliza o bot o
```

```
14                on_press: root.send_control_message(self.control_index)
15                image_on_bg_normal: 'images/buttons/g_on_b.png'
16                image_on_bg_down: 'images/buttons/g_off.png'
17                image_off_bg_normal: 'images/buttons/g_off.png'
18                image_off_bg_down: 'images/buttons/g_off_down.png'
19                control_index: 40
20                indication_index: 49
21            Label:
22                text: '[b]LOCAL[/b]'
23                markup: True  # Habilita o uso de tags de formata  o
24                halign:'center'
25                size_hint: (None, None)
26                size: (120, 30)
27                pos_hint: {'center_x': 0.5, 'center_y': 0.2}  #
                      Posiciona o label sobre o bot o
28                font_size: 14
29                height: 10  # Ajuste a altura conforme necess rio
30                color: (0, 0, 0, 1)
```

Algoritmo 3.18 – Definição de Botão em .kv

A definição do *widget* botão possui lógica similar a de um LED quando trata o *status* que monitora como ativado ou desativado, contudo possui mais imagens para implementar a animação de pressionar o botão. A principal diferença se encontra no fato de possuir endereço de indicação e também endereço de controle. O endereço de controle é utilizado no ato de pressionar o botão que chama o método send_control_message.

A seguir, o Algoritmo 3.19 apresenta a lógica para atualização de imagens de botões.

```
1     def update_all_button_backgrounds(self, new_status):
2         """Atualiza os planos de fundo de todos os bot es"""
3         for widget in self.walk():
4             if isinstance(widget, Widget) and hasattr(widget, 'is_button
                  ') and widget.is_button:
5                 if widget.indication_index == 0:
6                     continue
7
8                 index = cal_indication_index(widget.indication_index,
                      16)
9                 if len(new_status) > index:
10                    current_state = new_status[index]
11                    previous_state = self.button_states.get(widget, None
                          )
12                    if current_state != previous_state:
13                        if current_state == '1':
14                            widget.background_normal = widget.
                                  image_on_bg_normal # bg normal aceso
```

```
15                              widget.background_down = widget.
                                    image_on_bg_down # bg down aceso
16                          else:
17                              widget.background_normal = widget.
                                    image_off_bg_normal # bg normal apagado
18                              widget.background_down = widget.
                                    image_off_bg_down  # bg down apagado
```

Algoritmo 3.19 – Método para atualização dos Botões

3.5 Conexão da IHM com o *Driver*

A conexão da Interface Humano-Máquina com o *driver* Genisys será abordada através do envio das mensagens de controle e a interpretação das mensagens de indicação.

3.5.1 Envio de Mensagens de Controle

O envio de uma mensagem de controle se inicia com a interação de clique de algum botão na tela da IHM. No Algoritmo 3.20 observa-se que a ação de clique inicializa o método send_control_messages, e envia o control_index associado ao botão como parâmetro.

```
1     def send_control_message(self, control_index):
2         """Envia uma mensagem de controle com o valor digitado"""
3         try:
4             control_value = int(control_index)
5             self.driver.set_control_value(control_value)
6             self.ids.status_label.text = f"Controle enviado: {
                  control_value}"
7             logger.info(f"Controle enviado: {control_value}")
8         except ValueError:
9             self.ids.status_label.text = "Valor de controle inv lido"
10            logger.error("Valor de controle inv lido")
11        except Exception as e:
12            self.ids.status_label.text = f"Erro: {str(e)}"
13            logger.error(f"Erro ao enviar controle: {str(e)}")
```

Algoritmo 3.20 – Método que inicia e mensagem de controle

Esse método, por sua vez, chama o método set_control_value, e passa como parâmetro o valor do endereço como um inteiro, como observa-se no Algoritmo 3.21.

```
1     def set_control_value(self, value: int) -> None:
2         """Envia comando de controle para o dispositivo"""
3         if not 1 <= value <= self.max_control_value:
```

```
4             raise ValueError(f"Valor deve estar entre 1 e {self.
                  max_control_value}")
5
6         self.control_value = value
7         logger.info(f"Enviando valor de controle: {value}")
8         address = value-1 #para corrigir a contagem que inicia de 0 e
              n o de 1
9
10        # Gera as mensagens de controle
11        raw_message_on, hex_message_on = gerar_mensagem_controle(16, [
              address], False)
12        raw_message_off, hex_message_off = gerar_mensagem_controle(16, [
              address], True)
13
14        # Envia as mensagens de controle
15        if isinstance(raw_message_on, bytes) and isinstance(
              raw_message_off, bytes):
16            self.ser.write(raw_message_on)
17            logging.info(f"Mensagem de controle enviada: {hex_message_on
                  }")
18            time.sleep(0.5)
19            self.ser.write(raw_message_off)
20            logging.info(f"Mensagem de controle enviada: {
                  hex_message_off}")
21        else:
22            logger.error("Erro: gerar_mensagem_controle n o retornou
                  bytes")
```

Algoritmo 3.21 – Método que corrige o endereço e envia as mensagens

O método set_control_value é o que envia a mensagem através do pyserial. É importante destacar que uma mensagem de controle enviada sempre é seguida de um outro pacote de controle que retorna o valor enviado para o estado inicial, para que seja similar a um pulso. Antes do envio, as mensagens são criadas de acordo com o protocolo Genisys no método gerar_mensagem_controle, como observa-se no Algoritmo 3.22.

```
1  def gerar_mensagem_controle(num_words: int,
2                              control_addresses: List[int],
3                              reset: bool = False) -> Tuple[bytes, str]:
4      """Gera mensagem de controle para o protocolo Genisys.
5
6      Args:
7          num_words: N mero de palavras de controle
8          control_addresses: Lista de endere os de controle
9          reset: Flag para resetar o quarto byte
10
11     Returns:
```

```
        Tupla contendo (bytes da mensagem, string hexadecimal formatada)

    Raises:
        ValueError: Para endere os inv lidos
    """
    if not all(0 <= addr < num_words*8 for addr in control_addresses):
        raise ValueError("Endere os de controle inv lidos")
    control_bits = [0] * num_words * 8
    for addr in control_addresses:
        control_bits[addr] = 1


    message = bytearray([0xFC, 0x01])
    for i in range(num_words):
        chunk = control_bits[i*8:(i+1)*8]
        #print(f"\033[91m{chunk}\033[0m")
        if 1 in chunk:
            value = int(''.join(map(str, reversed(chunk))), 2)
            message.extend([i, value])

    if reset and len(message) >= 4:
        message[3] = 0x00


    raw_message, hex_message = build_complete_message(message)
    logger.debug(f"Mensagem de controle gerada: {hex_message}")
    return raw_message, hex_message
```

Algoritmo 3.22 – Método que gera os *bytes* de data

Nesta etapa o *payload* é criado, sendo este o que pode levar o *bit* associado para o valor lógico 1 ou retornar o valor para 0 na segunda chamada da função. Entretanto, a mensagem não é toda criada nesse método, pois ainda é necessário aplicar o CRC, o *escaping* e adicionar o terminador. A seguir, o Algoritmo 3.23 demonstra que todas as três etapas são tratadas na função build_complete_message.

```
def build_complete_message(packet: Union[str, bytes]) -> Tuple[bytes,
    str]:
    """Constr i mensagem completa com CRC e escaping.

    Args:
        packet: Pacote original em string hex ou bytes

    Returns:
        Tupla contendo (bytes da mensagem, string hexadecimal formatada)
    """
    if isinstance(packet, str):
```

```
11          data = bytes.fromhex(packet)
12      else:
13          data = packet
14
15      crc = calculate_crc16(data)
16      full_message = data + bytes([crc & 0xFF, (crc >> 8) & 0xFF,
            TERMINATOR])
17      escaped_message = apply_escaping(full_message)
18      return escaped_message, ' '.join(f"{b:02X}" for b in escaped_message
            )
```

Algoritmo 3.23 – Método que finaliza o pacote de mensagem

3.5.2 Interpretação das Mensagens de Indicação

A interpretação das mensagens de indicação se passa pelo processo de criação da *string* global "*status*", que é preenchida por binários, indicando se cada variável está em nível lógico 1 ou 0. Após o processamento inicial do *driver*, caso a mensagem tenha o *header* 0xF2, ela é tratada como uma mensagem de indicação, e passa pelo *handler* definido para ela, que está demonstrado no Algoritmo 3.24.

```
1       def _handle_indication(self, message: bytes) -> None:
2           """Handler para mensagens de indica  o (0xF2)"""
3           logger.info("Processando mensagem de indica  o...")
4
5           try:
6               unescaped = reverse_escaping(message)
7               binary_content = self._parse_indication_content(unescaped)
8               logger.info(f"Indication Binery Content: {binary_content}")
9
10              # Atualiza o status com o conte do bin rio
11              with self._lock:
12                  self.status = binary_content
13
14              with self._lock:
15                  self._safe_write(self.ACK_POLL_MESSAGE)
16
17          except Exception as e:
18              logger.error(f"Erro no processamento de indica  o: {str(e)
                }")
```

Algoritmo 3.24 – *Handler* para mensagens de Indicação

No método definido para o *handler* a mensagem tem o *escaping* removido e em seguida passa pelo processo de *parsing* para que a variável *status* seja atualizada.

Como apresentado nos métodos de atualização de imagens, a variável *status* é recorrentemente acessada para que a imagem seja atualizada de acordo com o endereço de indicação definido. Contudo, é necessária uma correção para que o endereço correto seja acessado. O Algoritmo 3.25 apresenta o método cal_indication_index que realiza a correção.

```
1  def cal_indication_index(indication: int, num_indication_words: int) ->
       int:
2      if 0 <= indication <= 8 * num_indication_words:
3          quociente = (indication - 1) // 8
4          resto = (indication - 1) % 8
5          quociente += 1
6          aux1 = quociente * 8
7          indication_index = aux1 - (resto + 1)
8      else:
9          indication_index = -1  # Defina um valor padr o ou lance uma
               exce   o
10         print("Indica   o definida fora do intervalo do mapa de bits")
11     return indication_index
```

Algoritmo 3.25 – Método para cálculo do endereço de indicação

O endereço configurado no mapa de *bits*, não é exatamente o mesmo endereço da *string* binária que possui a informação, pois ela é criada concatenando os 8bits de cada *bytes*, mas os bits são alocados da direita para a esquerda dentro de cada octeto. Portanto, é necessário fazer uma correção do valor configurado como endereço para que a posição correta seja acessada na *string "status"*. A correção está expressa no Algoritmo 3.25.

3.6 Comparativo Entre a Interface Desenvolvida e o Sistema Legado

A Tabela 1 apresenta alguns pontos que podem ser destacados na comparação entre o sistema legado e o proposto por este trabalho e cada um deles será tratado nas seções seguintes.

	Interface Desenvolvida	**Sistema Legado**
Desenvolvimento da Interface	Auxilio de IA	Fácil compreensão
Suporte de Linguagem	Ativo	Descontinuado
Acesso ao *Backend* da Solução	Possível	Apenas ao *Frontend*
Customização	Totalmente customizável	Customização limitada

Tabela 1 – Tabela Comparativa entre Sistema Desenvolvido e Sistema Legado

3.6.1 Desenvolvimento da Interface Gráfica

O desenvolvimento da interface gráfica no sistema legado é mais ágil e fácil de aprender para quem não teve nenhum contato com nenhuma das duas soluções, pois está associado à orientação a imagens, tornando o posicionamento das imagens na tela algo intuitivo. Já o sistema desenvolvido nesse trabalho possui um posicionamento baseado em coordenadas, que para quem não está familiarizado, pode ser custoso, mas possui a vantagem de ser mais customizável, por ter acesso a todo o desenvolvimento da tela e não somente a alguns objetos pré-setados. Além disso, o programa de edição do Adobe Flash não possui mais suporte, e o Kivy Language possibilita o auxilio de IA durante o desenvolvimento por ser totalmente em texto.

3.6.2 Suporte

A aplicação legado é toda desenvolvida pelo Adobe Flash, que foi descontinuado em 2020. Sendo assim, a aplicação que carrega as interfaces desenvolvidas e o executável Flash Portable que permite a edição de interfaces já não rodam em todas as máquinas atuais, tornando-se um problema para o desenvolvimento de novas interfaces com diferentes configurações de pátio.

3.6.3 Acesso ao *Backend*

O sistema legado atual, permite a personalização do *frontend*, entretanto não permite acesso ao *driver* de comunicação da aplicação principal. Portanto o sistema desenvolvido se torna superior ao permitir mudanças no *backend* da solução, tornando assim possível maior manutenção da aplicação ao longo dos anos.

3.6.4 Customização

Os pontos anteriores colaboram para a conclusão de que a customização da nova aplicação seja superior à aplicação legado. Permitindo que sejam feitos programas robustos, com interfaces modernas e customizadas para cada cliente diferente.

4 Resultados

Este capítulo apresenta a solução final e realiza os testes necessários para validar o sistema desenvolvido. Através da comparação dos *status* apresentados no supervisório e seu estado atual que pode ser obtido no SATS, é possível validar que a mensagem enviada pelo CLP está sendo corretamente interpretada e transformada em algo visual para quem operar o programa. Da mesma forma, é possível testar o envio das mensagens de controle, observando se no clique do botão da IHM o botão correspondente no SATS também é ativado.

4.1 Resultados da Decodificação de Mensagens

A decodificação das mensagens é de suma importância para que o supervisório apresente o verdadeiro *status* atual das variáveis monitoradas. Portanto, a principal mensagem a ser decodificada é a de Indicação (0xF2). Para fazer a validação do que tem sido apresentado na interface do programa, é necessário comparar lado a lado com o que o SATS apresenta, tanto visualmente quanto o valor lógico da variável em questão.

O primeiro teste é observar se, quando estabelecida a conexão o estado inicial do PCL condiz com o estado apresentado pelo simulador.

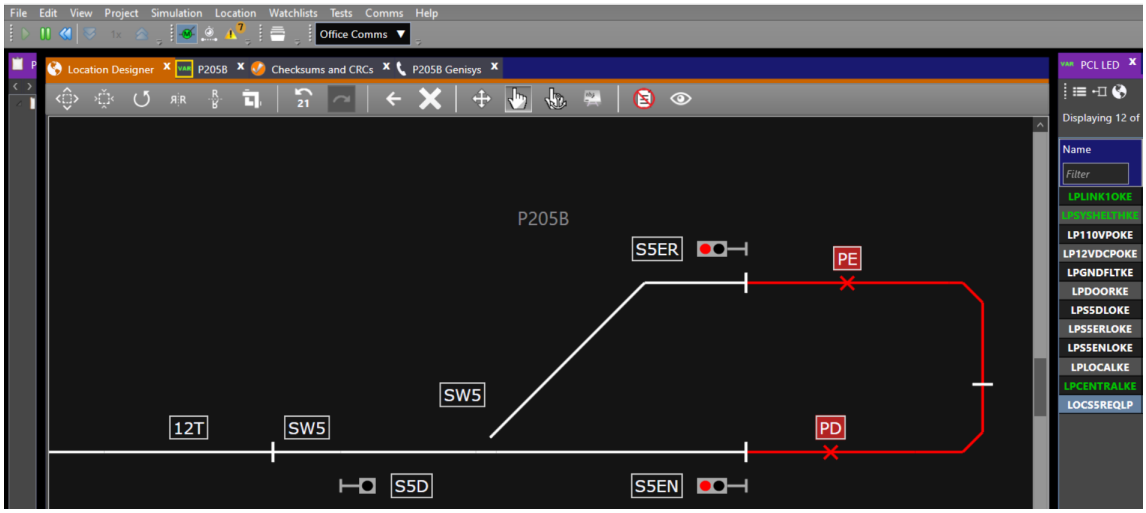

Figura 16 – Estado Inicial do SATS

Fonte: Autor

A Figura 16 apresenta graficamente máquina de chave em normal, sem nenhuma ocupação dos circuitos de via 12T e W5T, nenhuma interdição de chave ou de linha. A tabela na extremidade direita da imagem apresenta o nível lógico de algumas variáveis monitoradas pelo PCL, entre eles, os alarmes, a saúde do sistema, o *link* com a locação adjacente e o modo de controle estabelecido. Sendo assim é possível observar que nenhum

alarme está ativado, a saúde e o *link* estão estabelecidos e o modo de controle atual é central. Dessa maneira, a Figura 17 nos mostra que a interface do programa desenvolvido está condizente com o que está sendo simulado no SATS.

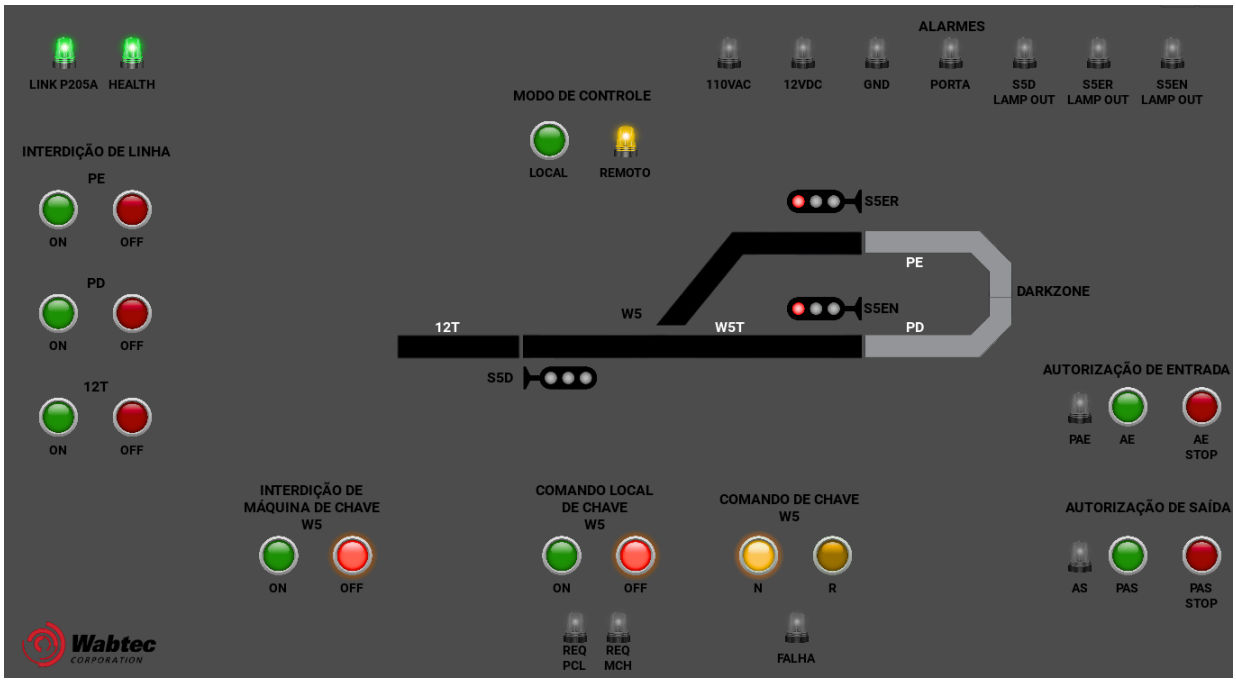

Figura 17 – Estado Inicial do PCL

Fonte: Autor

Para verificar se o PCL está acompanhando o que é simulado no SATS, através da interpretação correta das mensagens, diversos estados de variáveis serão alterados, a fim de abranger diferentes tipos de imagens criada no PCL.

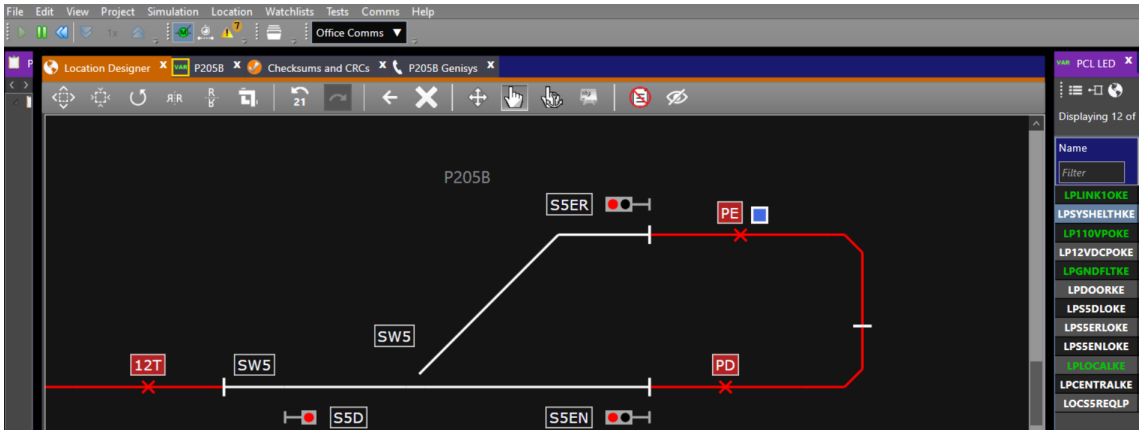

Figura 18 – Segundo Estado do SATS

Fonte: Autor

Na Figura 18 observa-se uma interdição de via representada pelo quadrado azul, uma ocupação do circuito 12T, e a alteração do estado dos alarmes de falha 12VDC e

falha do GND. Além disso, o sinal S5D passou a apresentar aspecto vermelho, devido a ocupação do circuito 12T.

Na medida em que o simulador alterou alguns *status* de variáveis, o PCL acompanhou as mudanças, sendo possível observar na Figura 19 que os LED's de alarmes acenderam, o LED de saúde apagou, o botão de interdição de linha PE também acendeu, o circuito 12T passou a ser ocupado e o sinal S5D acendeu.

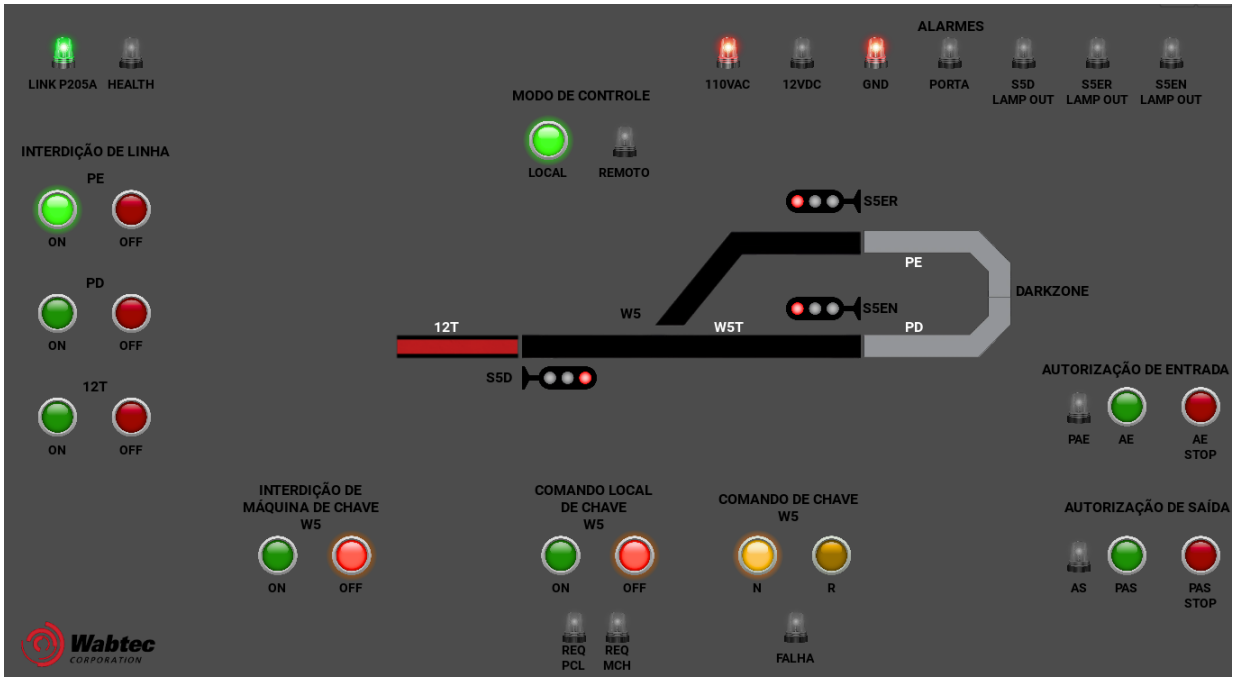

Figura 19 – Segundo Estado do PCL

Fonte: Autor

Concluindo a decodificação de mensagem, é importante validar que a nova solução é capaz de enviar mensagens de controle para o CLP. Seguindo o último estado da simulação apresentado neste trabalho, o modo de controle local já está estabelecido, portanto é possível enviar mensagens de controle que alterem o *status* atual da simulação. Para esse teste busca-se movimentar a máquina de chave para reverso.

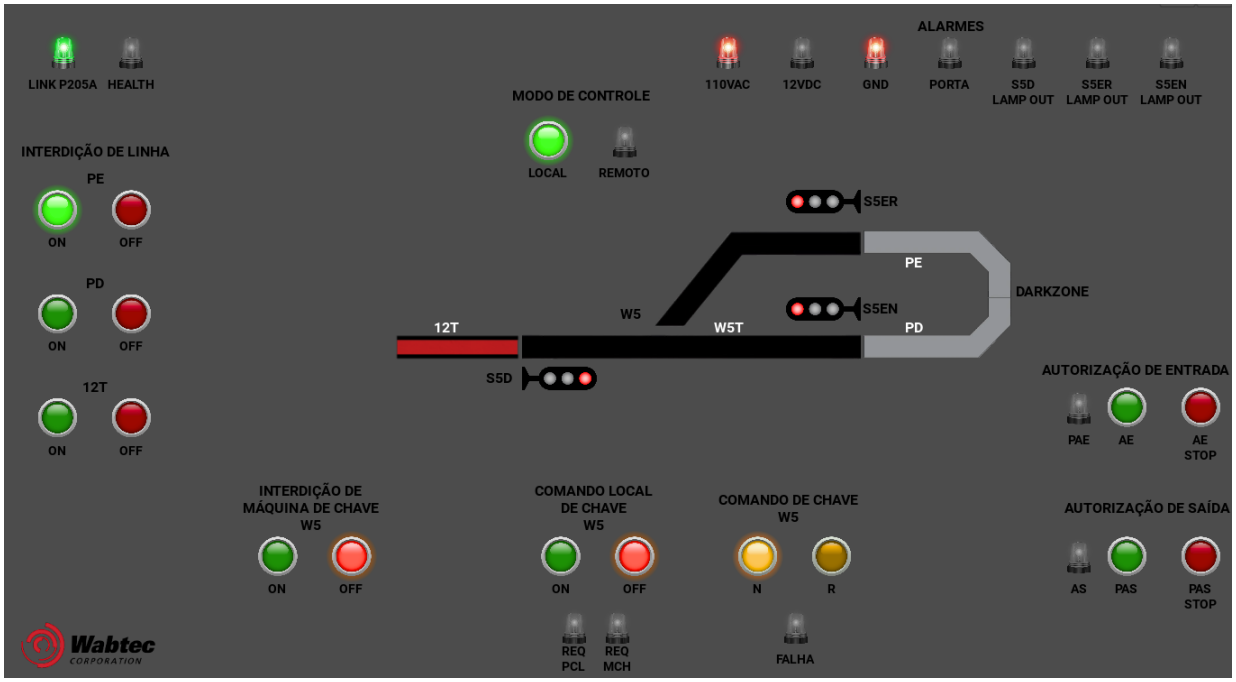

Figura 20 – Controle recebido pelo SATS

Fonte: Autor

Na Figura 20, é possível observar que após o clique no botão amarelo "R", de comando de máquina de chave, a variável de comando da MCH foi ativada e a contagem de tempo para que ela feche o curso na posição de reverso foi iniciada, podendo concluir que a mensagem em Genisys está sendo criada corretamente e seguindo o protocolo, ela consegue enviar comando para o CLP.

4.2 Avaliação da Interface Gráfica

O sistema legado possui três interfaces que são utilizadas na aplicação. A primeira, Figura 21, permite escolher qual o modelo de pátio será carregado, as opções irão variar de acordo com os arquivos de extensão `.swf` e .xml carregados na pasta do programa. Esses arquivos são gerados previamente através do editor de interfaces do Adobe Flash Portable.

Figura 21 – Primeira Interface PCL Legado

Fonte: Autor

A segunda interface do sistema, Figura 22, legado deveria atuar como uma tela de *login*. Entretanto, essa funcionalidade não está mais presente no programa, não sendo necessária nenhuma senha para acessar a terceira interface.

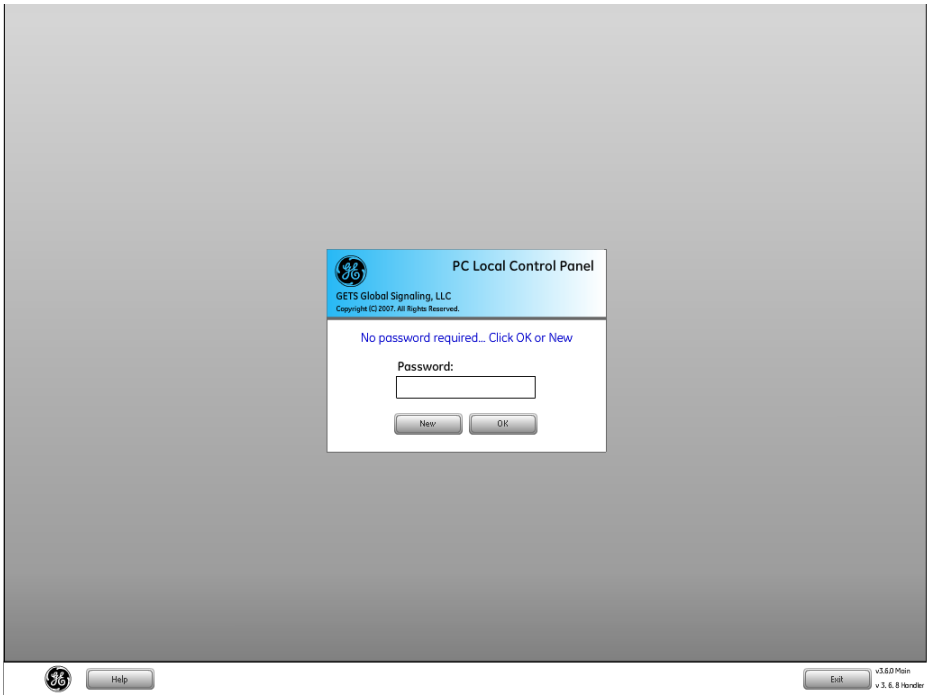

Figura 22 – Segunda Interface PCL Legado

Fonte: Autor

A terceira e última interface é a que de fato o supervisório está presente, sendo possível monitorar o estado dos ativos de campo e alarmes pré-definidos e também atuar como um painel de controle local ao enviar comandos através dos botões da interface.

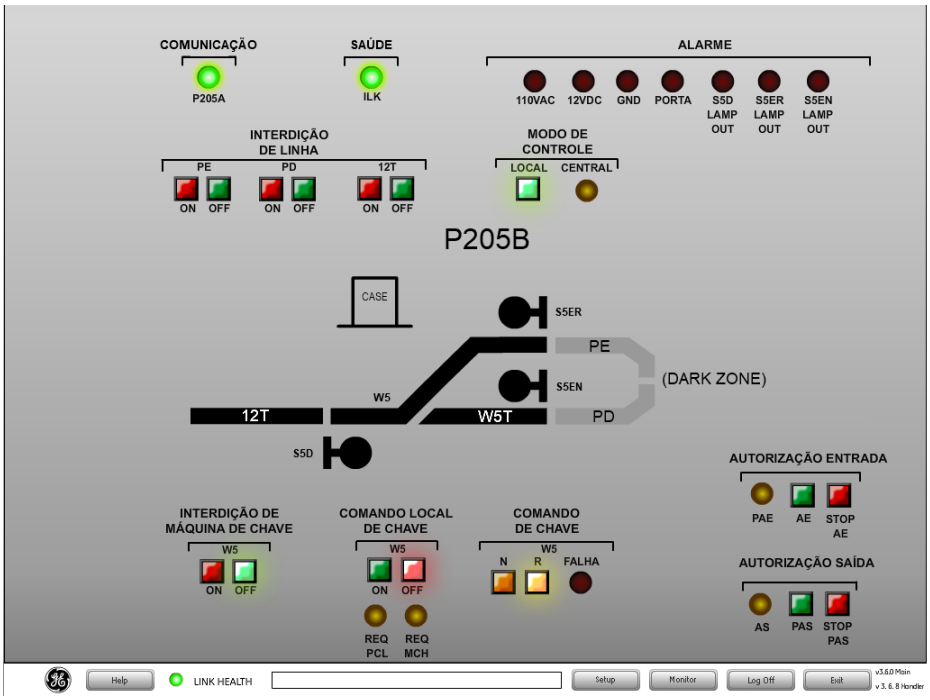

Figura 23 – Terceira Interface PCL Legado

Fonte: Autor

A Figura 23 mostra que na barra inferior dessa interface é possível acessar as configurações de comunicação, onde é definido se a comunicação será por *User Dataprogram Protocol/Internet Protocol* (UDP/IP) ou serial, a porta COM, o protocolo de comunicação e algumas outras configurações secundárias apresentadas na Figura 24.

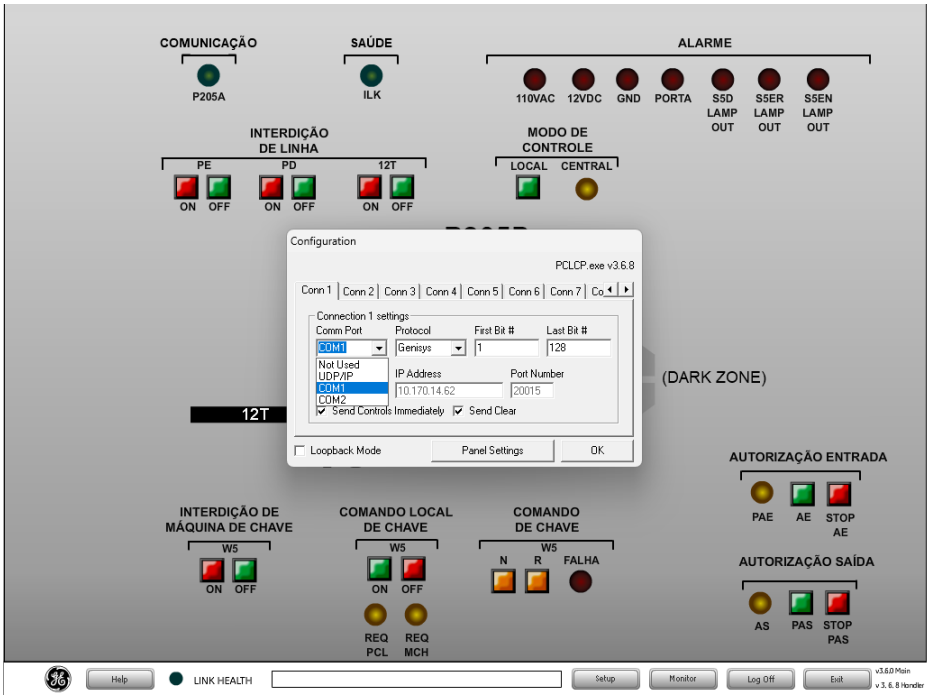

Figura 24 – Configurações de conexão PCL Legado

Fonte: Autor

É possível também acessar o monitor serial da aplicação, a Figura 25 evidencia a troca de mensagens do painel com o CLP.

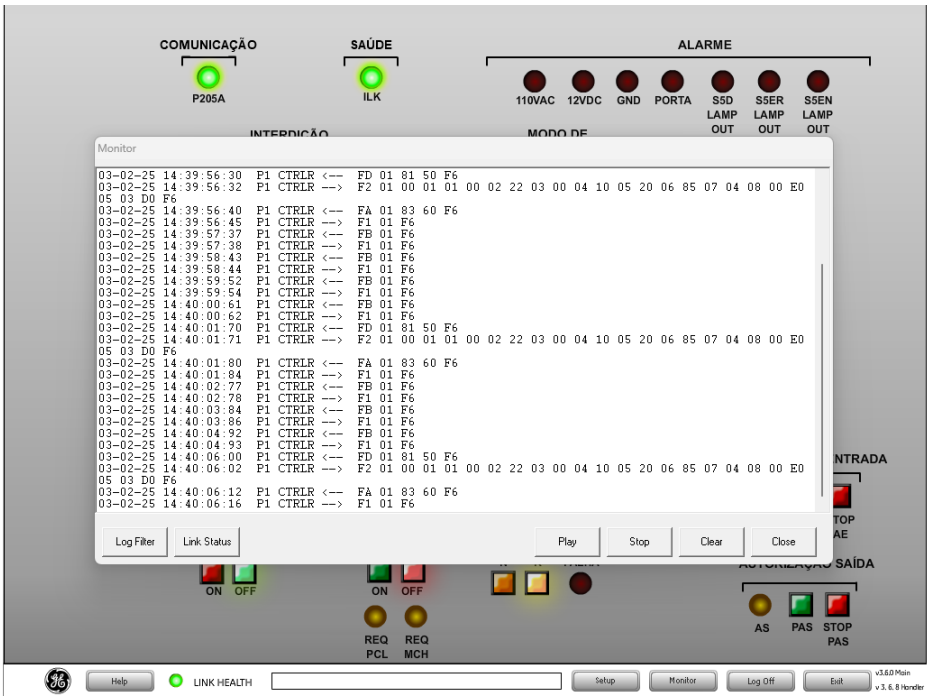

Figura 25 – Monitor PCL Legado

Fonte: Autor

A nova interface gráfica, como pode ser visto na Figura 26, apresenta algumas diferenças estéticas em relação ao sistema legado, apresentando um *layout* mais moderno, e configurações de conexão na primeira tela.

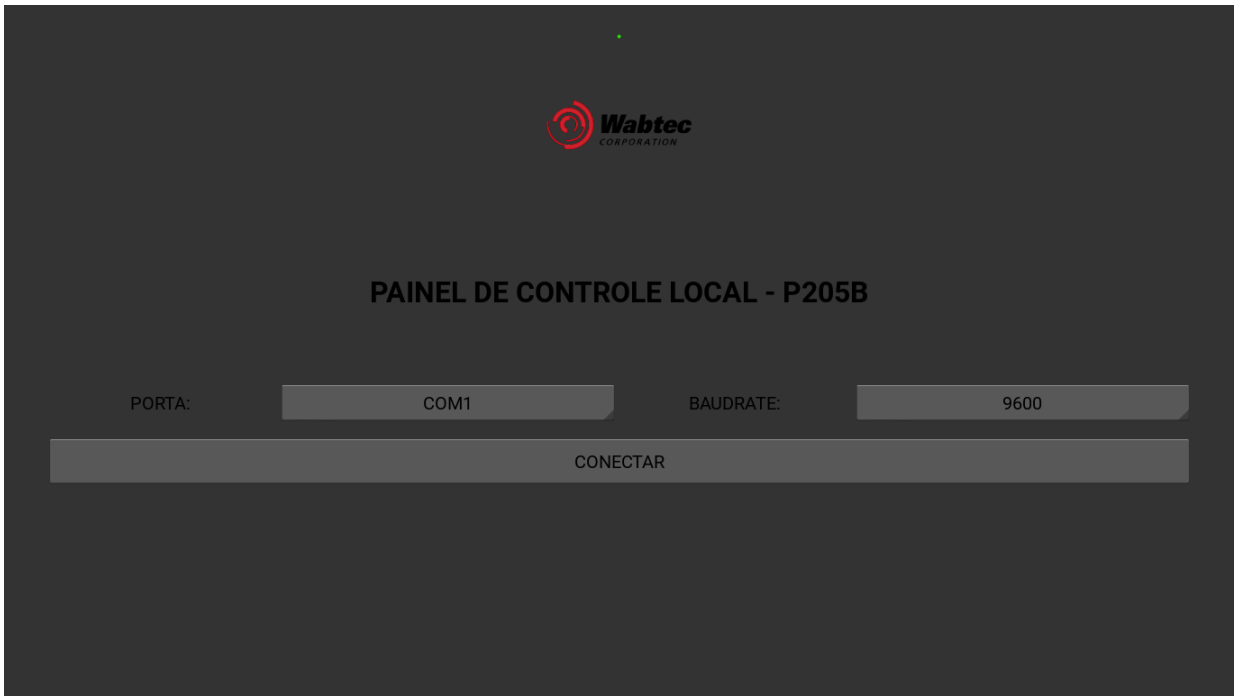

Figura 26 – Interface de Conexão PCL

Fonte: Autor

A segunda interface gráfica, diferente do que ocorre no PCL legado, já apresenta o supervisório, e só é carregada caso a conexão com o CLP seja estabelecida. A Figura 27 apresenta a interface do supervisório desenvolvida neste trabalho.

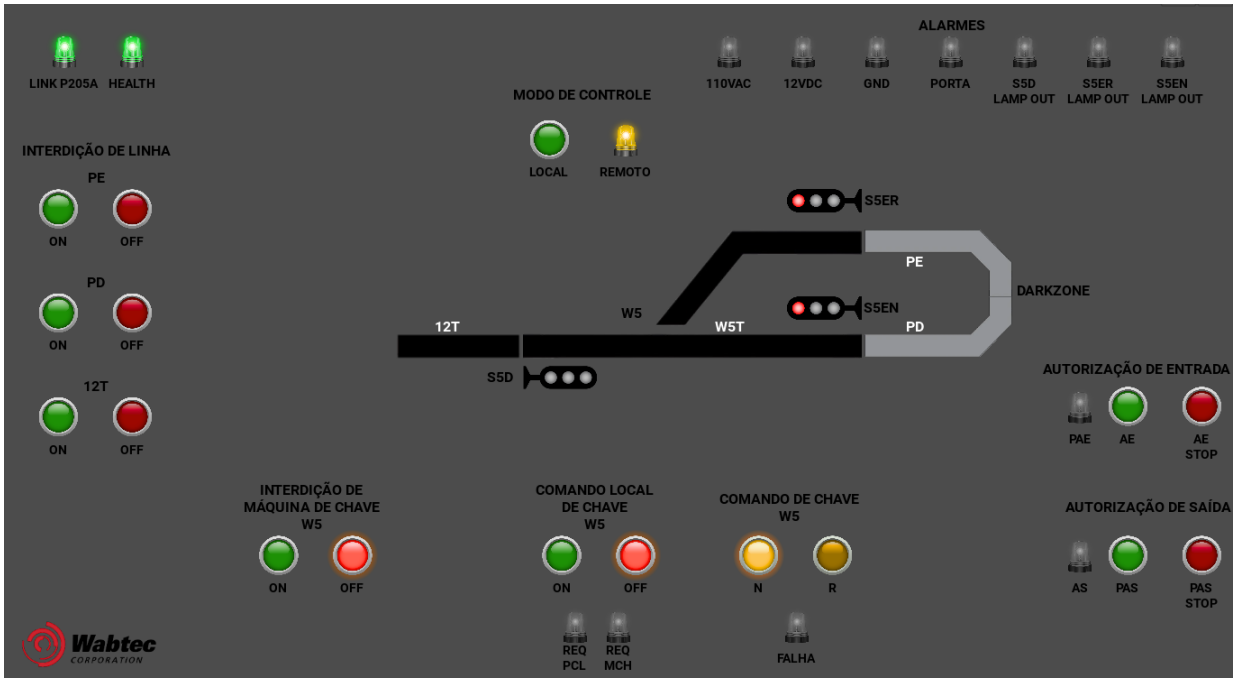

Figura 27 – Interface de Conexão PCL

Fonte: Autor

Ademais, algumas funcionalidades não foram implementadas. A primeira é a impossibilidade de acompanhar o monitor serial na própria aplicação, sendo necessário acessar o monitor através do VSCode. A segunda, se trata de uma nova abordagem para o conceito do sistema. A aplicação legado carrega vários *layouts* pré-definidos em uma pasta do programa, já a aplicação deste trabalho propõe um arquivo executável para cada configuração de pátio.

4.3 Desempenho e Usabilidade.

A IHM apresenta desempenho satisfatório quanto a responsividade dos botões, enviando as mensagens de controle de maneira instantânea. Porém, a responsividade visual ainda é inferior ao sistema legado. Quanto a atuação como supervisório, com indicações em estado constante, a aplicação é consistente e não apresenta nenhuma incongruência. Contudo, no momento em que algum *status* é alterado, a atualização da interface apresenta falsas indicações, demorando cerca de 2 segundos até normalizar e corrigir a interface, o que é um ponto de atenção que deve ser tratado para garantir segurança ao utilizar o sistema.

A Tabela 2, apresenta um comparativo entre as funcionalidades presentes no sistema desenvolvido e no sistema legado. Observa-se que o sistema legado possui mais funcionalidades que a nova IHM, o que limita sua usabilidade do novo sistema para cenários além dos simulados neste trabalho.

	Interface Desenvolvida	**Sistema Legado**
Supervisão de *Status*	Sim	Sim
Envio de Comandos	Sim	Sim
Comunicação Serial	Sim	Sim
Comunicação TCP/IP	Não	Sim
Monitor Serial	Não	Sim

Tabela 2 – Tabela Comparativa de Desempenho e Usabilidade

5 Conclusão e Propostas Para Trabalhos Futuros

Este capítulo apresenta as considerações finais do trabalho de substituição de um sistema baseado em tecnologias obsoletas.

5.1 Contribuições do Trabalho

O desenvolvimento deste trabalho resultou em avanços na modernização de sistemas supervisórios de controle ferroviário, validando a viabilidade de substituir tecnologias legadas por soluções abertas e adaptáveis. A integração entre o *driver* Genisys e a Interface Humano-Máquina em Kivy demonstrou ser uma alternativa eficiente, mesmo com recursos limitados.

A principal contribuição é destacada na implementação do *driver* Genisys, que permitiu decodificar e validar a estrutura de *frames* do protocolo, garantindo uma comunicação confiável com o CLP por meio de mecanismos de CRC e *escaping* de dados. A IHM desenvolvida, por sua vez, atende aos requisitos para substituir o sistema legado baseado em Adobe Flash, oferecendo uma aplicação personalizável e de fácil instalação por ser desenvolvida inteiramente em Python, renovando o acesso e a vida útil da solução.

Testes realizados com o simulador SATS comprovaram a eficácia do sistema em interpretar mensagens de indicação e transformá-las em dados visuais, e enviar comandos de controle. A solução também introduziu maior transparência no desenvolvimento, por permitir acesso ao *backend*, facilitando futuras customizações e melhorias.

5.2 Propostas Para Trabalhos Futuros

Esta seção apresenta melhorias que podem ser implementadas à atual solução, a fim de obter melhores resultados e expandir a aplicabilidade do sistema.

Considerando as atuais limitações de desempenho do produto, é necessário uma revisão do código base para encontrar possíveis otimizações da atualização da interface, melhorando a resposta visual do programa e a confiabilidade do que é apresentado. Além disso, integrar um módulo de geração de *logs* diretamente na interface, permitirá um diagnóstico de falhas e auditoria de comandos enviados.

As funcionalidades do sistema legado também podem ser implementadas caso seja vantajoso para a escalabilidade do produto. O desenvolvimento da comunicação TCP/IP como uma alternativa à serial, será necessário para aumentar a aplicabilidade em que a porta serial do CLP não esteja disponível. Além disso, implementar uma ferramenta de carregamento dinâmico de *layouts* de pátio, permitindo que usuários importem configurações sem necessidade de interpretar o código todo novamente.

Testes em ambientes reais, com equipamentos ferroviários, tornam-se de suma importância para que a confiabilidade do produto seja atestada. Portanto, é a comunicação com o CLP(ElectroLogIXS) em laboratórios e em ambientes de campo, quando as condições podem se tornar adversas, que validarão a robustez do sistema.

5.3 Considerações Finais

Este trabalho estabeleceu as bases para a compreensão e utilização do *driver* Genisys não só para o painel de controle local em questão, mas também para outras aplicações que possam utilizar o protocolo que está presente no ambiente ferroviário.

As propostas futuras visam transformar o produto em uma solução industrialmente relevante, sendo capaz de substituir integralmente o sistema legado atual, eliminando a dependência de tecnologias obsoletas.

REFERÊNCIAS

1 AREMA. *American Railway Engineering and Maintenance-of-Way Association Communications and Signal Manual of Recommended Practice*. [S.l.], 2021.

2 KEELER, B. *Genisys Protocol Dissector for Wireshark*. 2009. <https://www.wireshark.org>. Acesso em: 10 out. 2024.

3 ICSNPP. *ICSNPP-Genisys: Zeek Plugin for Genisys Protocol*. 2023. <https://github.com/cisagov/ICSNPP>. Acesso em: 13 out. 2024.

4 SIGNAL, U. S. . *SM 6700A: GENISYS Serial Communication Protocol*. [S.l.], 1996. Technical Manual.

5 VERZYNOV, S. N.; BOCHKAREV, I. V.; KHRAMSHIN, V. R. Development of line locator software component for mobile operating systems. In: *2020 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM)*. [S.l.: s.n.], 2020. p. 1–5.