UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA MECÂNICA

LUCAS FARIAS NOGUEIRA

Aplicativo para controle de velocidade angular utilizando MQTT como protocolo de comunicação

Uberlândia
2025

LUCAS FARIAS NOGUEIRA

Aplicativo para controle de velocidade angular utilizando MQTT como protocolo de comunicação

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia Mecatrônica da Universidade Federal de Uberlândia como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica

Área de concentração: Engenharia Mecatrônica

Orientador: José Jean-Paul Zanlucchi de Souza Tavares

Uberlândia

2025

LUCAS FARIAS NOGUEIRA

Aplicativo para controle de velocidade angular utilizando MQTT como protocolo de comunicação

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia Mecatrônica da Universidade Federal de Uberlândia como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica

Área de concentração: Engenharia Mecatrônica

Uberlândia, 26/09/2025

Banca Examinadora:

_____

Prof. Dr. José Jean-Paul Zanlucchi de Souza Tavares (UFU)

_____

Prof. Ms. Werley Rocherter Borges Ferreira (UFU)

_____

Prof. Ms. Nei Oliveira de Souza (IFTM)

Dedico este trabalho a minha mãe, e aos meus irmãos pelo estímulo, carinho e compreensão.

AGRADECIMENTOS

Agradeço ao professor e amigo Prof. Dr. José Jean-Paul Zanlucchi de Souza Tavares o incentivo, motivação e orientação nesta caminhada acadêmica.

Aos colegas Eduardo Apolinário Lopes, Guilherme Gonzaga Silva, Pedro Afonso Silva por dividir momentos e apoios do início ao fim da graduação.

Em especial, ao colega João Marcos Souza Ferreira, que fez parte muito relevante do desenvolvimento do software.

“Qualquer tecnologia suficientemente
avançada é indistinguível da magia.”
(Arthur C. Clarke, 1973)

RESUMO

Em tempos de experienciação de eventos extraordinários, como pandemias, guerras e mudanças radicais em nosso planeta temos a necessidade de adaptação cada vez mais rápido e otimizada. Uma dessas adaptações necessárias, podemos citar as aulas práticas de Controle Linear da Universidade Federal de Uberlândia que são ministradas no Laboratório de Ensino para Mecatrônica, aulas essas essenciais para a aplicação dos conhecimentos teóricos adquiridos durante o curso. Para minimizar o impacto aos discentes durante períodos de estabilidade social como os ocorridos durante a pandemia da COVID-19, pode ser idealizado uma solução disseminada pela Industria 4.0. Utilizando uma arquitetura da *Industrial Internet of Things* (IIoT) que nos permite conexão de diversos dispositivos, sensores e máquinas em tempo real, remotamente, gerando rentabilidade e confiabilidade em diversas situações. Aplicado juntamente a outras ferramentas como o protocolo *Message Queuing Telemetry Transport* (MQTT) permite a comunicação de dispositivos conectados à rede. Considerando tais ferramentas como focais, esse projeto apresenta uma solução que permite a realização de aulas remotas de todo o módulo prático da disciplina controlando os experimentos por um aplicativo de celular, este por sua vez altamente adaptável para outros ambientes, diferentes do meio acadêmico. Complementando essa arquitetura, temos microcontroladores ESP32 para controle dos protótipos práticos e comunicação com o *broker* local MQTT e aplicativo *mobile* para dispositivos móveis.

Palavras-chave: IIoT; MQTT; ESP32; Mecatrônica; Aula Remota de Controle

ABSTRACT

In times of extraordinary events, such as pandemics, wars, and radical changes on our planet, we face the need for increasingly rapid and optimized adaptation. One of these necessary adaptations can be seen in the practical Linear Control classes at the Federal University of Uberlandia, held in the Mechatronics Teaching Laboratory, which are essential for applying the theoretical knowledge acquired during the course. To minimize the impact on students during periods of social instability, such as those that occurred during the COVID-19 pandemic, a solution disseminated by Industry 4.0 can be conceived. Utilizing an Industrial Internet of Things (IIoT) architecture, which allows us to connect various devices, sensors, and machines in real-time and remotely, generates profitability and reliability in diverse situations. Applied together with other tools like the Message Queuing Telemetry Transport (MQTT) protocol, it enables communication between network-connected devices. Considering these tools as focal points, this project presents a solution that allows for the remote execution of the entire practical module of the discipline by controlling experiments through a mobile application, which is, in turn, highly adaptable to other environments outside the academic sphere. Complementing this architecture, we have ESP32 microcontrollers for controlling the practical prototypes and communicating with the local MQTT broker and the mobile application.

Keywords: IIoT; MQTT; ESP32; Mechatronics; Control Remote Class.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACK	*Acknowledgment* (Confirmação de recebimento)
ADC	Conversores Analógico-Digitais
AOT	*Ahead-of-Time* (Compilação antecipada)
CC	Corrente Contínua
DAC	Conversores Digital-Analógicos
ESP32	Microcontrolador da Espressif Systems com Wi-Fi e *Bluetooth* integrados
GCC	GNU *Compiler Collection* (Coleção de Compiladores GNU)
GPIOs	*General Purpose Input/Output* (Entrada/Saída de Propósito Geral)
I2C	*Inter-Integrated Circuit* (Interface de comunicação serial)
IDE	*Integrated Development Environment* (Ambiente de Desenvolvimento Integrado)
IHM	Interface Homem-Máquina
IIoT	*Industrial Internet of Things* (Internet Industrial das Coisas)
IoT	*Internet of Things* (Internet das Coisas)
iOS	Sistema operacional móvel da *Apple Inc*.
JIT	*Just-in-Time* (Compilação em tempo de execução)
LE	*Low Energy* (Baixo consumo de energia)
LEM	Laboratório de Ensino para Mecatrônica
M2M	*Machine-to-Machine* (Comunicação máquina a máquina)
MA	Malha Aberta
MAPL	Laboratório de Planejamento Automático de Manufatura
MF	Malha Fechada
MQTT	*Message Queuing Telemetry Transport*
P	Proporcional
PI	Proporcional-Integral
PUBACK	*Publish Acknowledgment*
PWM	*Pulse Width Modulation*
QoS	*Quality of Service*
RPM	Rotações por Minuto
SPI	*Serial Peripheral Interface*
TCP/IP	*Transmission Control Protocol/Internet Protocol*

TLS	*Transport Layer Security*
UART	*Universal Asynchronous Receiver-Transmitter*
UFU	Universidade Federal de Uberlândia
UI	*User Interface* (Interface de Usuário)
VS Code	*Visual Studio Code*

SUMÁRIO

1 INTRODUÇÃO

Vivemos um momento de intensas e rápidas transformações, impulsionadas pelo desenvolvimento das comunicações e pela redução de custos dos microprocessadores, o que caracteriza a nova revolução das técnicas, conhecida como Indústria 4.0. Esta nova era é marcada pela automação e pela integração de tecnologias inteligentes e conectadas, onde a Internet das Coisas (IoT) permite a comunicação entre máquinas, equipamentos e dispositivos, criando ambientes colaborativos e descentralizados. Contudo, eventos extraordinários, como a pandemia de COVID-19, evidenciaram a necessidade de modelos mais flexíveis e resilientes, especialmente em áreas que dependem de atividades presenciais.

Nesse cenário, o setor educacional enfrentou desafios significativos, principalmente na condução de aulas práticas que são essenciais para a formação de engenheiros. Disciplinas como Controle Linear, ministradas na Universidade Federal de Uberlândia, que dependem do uso de bancadas e equipamentos no Laboratório de Ensino para Mecatrônica (LEM), foram diretamente impactadas. A impossibilidade de acesso físico aos laboratórios comprometeu o aprendizado, uma vez que simulações em software, embora úteis, não conseguem reproduzir integralmente as nuances e desafios da experimentação prática. Essa lacuna demonstrou a urgência em desenvolver soluções que permitam a continuidade das atividades práticas de forma remota.

Para endereçar esse problema, este projeto propõe uma solução baseada na arquitetura da Internet Industrial das Coisas (IIoT), um dos pilares da Indústria 4.0, que viabiliza a conexão remota e em tempo real de dispositivos, sensores e máquinas. O elemento central desta arquitetura é o protocolo de comunicação *Message Queuing Telemetry Transport* (MQTT), que se destaca por ser leve, eficiente e robusto, tornando-se ideal para a comunicação entre dispositivos com recursos limitados, como os microcontroladores ESP32. A solução completa envolve o controle dos experimentos por meio de um aplicativo móvel, que atuará como Interface Homem-Máquina (IHM), permitindo que os alunos operem os protótipos de qualquer local.

A justificativa para a execução deste trabalho reside não apenas na necessidade de contornar as limitações impostas por períodos de instabilidade, mas também na modernização do próprio processo de ensino-aprendizagem. Ao integrar conceitos de IIoT, comunicação M2M (*Machine-to-Machine*) e controle remoto, o projeto oferece uma ferramenta pedagógica poderosa, alinhada às competências exigidas do engenheiro mecatrônico no mercado atual. A implementação de um sistema de controle remoto para as aulas práticas de Controle Linear qualifica os estudantes com tecnologias de ponta e abre precedentes para a aplicação em outras disciplinas e ambientes, para além do meio acadêmico.

1.1 Objetivos

1.1.1 Objetivo principal

Desenvolver um aplicativo educacional voltado ao ensino de controle linear de posição e velocidade de um motor utilizando MQTT como protocolo de comunicação, permitindo a

integração entre conceitos teóricos e práticas experimentais, de forma acessível e compatível com os princípios da Indústria 4.0.

1.1.2 Objetivos específicos

Para tal, os objetivos específicos do trabalho englobam o estudo e a implementação de um fluxo de comunicação bidirecional baseado no protocolo MQTT, que será gerenciado por um *broker* local para interligar os microcontroladores ESP32 a um aplicativo de controle. Este aplicativo será desenvolvido com o *framework Flutter* e a linguagem *Dart*, priorizando uma interface intuitiva que permita ao aluno inserir parâmetros e operar os experimentos de forma didática. Adicionalmente, busca-se programar os microcontroladores ESP32 em linguagem C para que atuem como a interface de controle dos protótipos físicos, decodificando os comandos recebidos via MQTT e retransmitindo os dados coletados pelos sensores. Por fim, o projeto visa validar a arquitetura completa através da sua aplicação direta nos roteiros de aula da disciplina, demonstrando na prática sua eficácia como uma ferramenta para o ensino remoto e como meio de aplicação aos conceitos da Indústria 4.0.

1.2 Justificativa

A relevância deste trabalho se manifesta em duas frentes principais: a tecnológica e a pedagógica. No âmbito tecnológico, o projeto alinha-se diretamente aos princípios da Indústria 4.0, aplicando conceitos de Internet das Coisas Industrial (IIoT) em um cenário prático. A criação de um sistema de controle remoto, que integra hardware embarcado, um aplicativo móvel e um protocolo de comunicação como o MQTT, demonstra a viabilidade de desenvolver soluções de automação flexíveis, escaláveis e de baixo custo. Do ponto de vista pedagógico, o projeto aborda uma necessidade crítica evidenciada por cenários de ensino a distância: a continuidade das atividades práticas laboratoriais, que são indispensáveis para a consolidação do conhecimento em engenharia.

A concretização deste trabalho foi possível graças à aplicação de um conjunto de conhecimentos multidisciplinares adquiridos ao longo da graduação em Engenharia Mecatrônica. A disciplina de Controle de Sistemas Lineares serviu como pilar central, fornecendo toda a base teórica sobre o comportamento de sistemas em malha aberta e malha fechada, além dos fundamentos para a implementação do controlador proporcional, que é o foco do experimento prático abordado.

Os conhecimentos de Eletrônica Básica e Eletrônica Digital foram cruciais para a correta interface entre os componentes de hardware, como o microcontrolador ESP32, o driver de motor Ponte H L298N e o *encoder*. A disciplina de Sistemas Digitais proveu as competências específicas para a programação do firmware do ESP32, incluindo o mapeamento de pinos, a geração de sinais PWM (Modulação por Largura de Pulso) e a leitura de sensores.

A base de software foi consolidada pela disciplina de Algoritmos e Programação de Computadores, que fundamentou a lógica de programação utilizada tanto no firmware em C/C++ quanto no aplicativo em Dart. O desenvolvimento da comunicação em rede foi sustentado pelos conceitos de Redes Industriais, essenciais para a compreensão da arquitetura cliente-servidor, do protocolo TCP/IP (base do MQTT) e do gerenciamento de conexões em rede local. Por fim, a construção de uma interface intuitiva e funcional no aplicativo móvel se beneficiou de princípios de Engenharia de Software, visando uma experiência de usuário clara e eficaz.

Para finalizar todo esse conjunto de conceitos aprendidos durante o curso, a disciplina de Internet das Coisas Industriais (IIoT), foi crucial para nos dar o vislumbre de como harmonizar tudo em uma solução real e usual para os problemas encontrados no dia a dia, realizando a integração homem máquina.

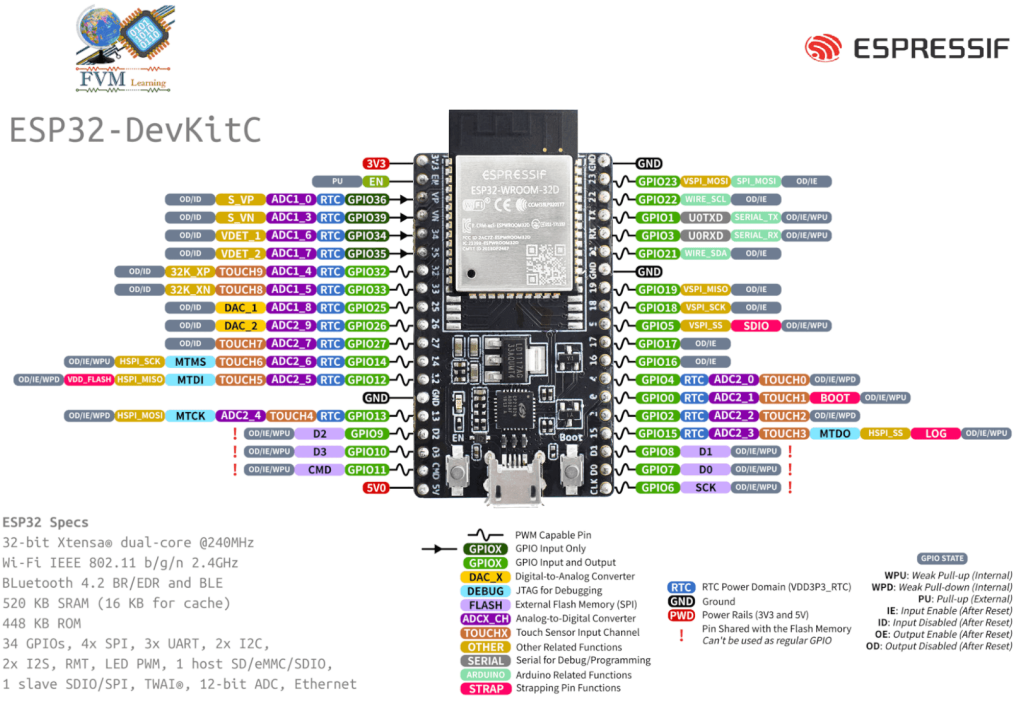
Este documento está estruturado da seguinte forma para apresentar o desenvolvimento e os resultados do projeto: o Capítulo 1 detalha a introdução, a contextualização do problema e os objetivos. O Capítulo 2 apresenta a revisão bibliográfica, abordando os conceitos fundamentais sobre as tecnologias empregadas. O Capítulo 3 descreve a metodologia, detalhando o desenvolvimento do firmware para o microcontrolador e do aplicativo móvel. O Capítulo 4 apresenta os resultados práticos obtidos com a implementação do sistema. Finalmente, o Capítulo 5 traz as considerações finais, resumindo as contribuições do trabalho e sugerindo propostas para desenvolvimentos futuros.

2 REVISÃO BIBLIOGRÁFICA

2.1 Microcontrolador ESP32

O ESP32 é uma placa micro controladora de baixo custo, em torno de R$40,00, desenvolvida pela empresa *Espressif Systems*, projetada para ser uma solução acessível e eficiente para projetos de automação e robótica, especialmente no campo da Internet das Coisas (IoT). Sua arquitetura é baseada em um microprocessador *Xtensa® dual-core* de 32 *bits*, que pode operar em até 240 MHz, e já integra módulos para comunicação sem fio, o que representa uma grande vantagem em relação a outros microcontroladores. Na Figura 1 vemos a descriminação dos pinos existentes no ESP32 e especificações de suas entradas e saídas.

Figura 1 - Pinagem ESP32



Fonte: https://www.fvml.com.br/2022/04/pinagem-pinout-esp32-devkitc.html

Uma das características mais notáveis do ESP32 é sua ampla conectividade, oferecendo suporte nativo a *Wi-Fi* (2.4 GHz), *Bluetooth* v4.2 e *Bluetooth Low Energy* (LE). Essa versatilidade o torna ideal para conectar dispositivos remotos com baixo consumo de rede. Além disso, o componente possui um design robusto, sendo capaz de operar de forma confiável em ambientes industriais, com uma faixa de temperatura operacional de -40°C a +125°C.

O *chip* foi projetado com foco em baixo consumo de energia, incorporando recursos como *clock gating*, múltiplos modos de energia e dimensionamento dinâmico de potência, o que o torna adequado para dispositivos móveis e aplicações de IoT.

Para a programação, o ESP32 é altamente flexível. Ele pode ser programado em linguagens como C/C++ através da IDE do Arduino, uma plataforma popular devido à sua vasta comunidade e facilidade de uso. O ambiente de desenvolvimento oficial do fabricante é o *Espressif IoT Development Framework* (ESP-IDF), baseado no *FreeRTOS*®, que utiliza *CMake* para a organização do processo de compilação e é compatível com diversas linguagens suportadas pelo compilador GCC, como C e C++. Do ponto de vista de hardware, a placa de desenvolvimento oferece uma variedade de periféricos, incluindo até 26 pinos de entrada/saída de propósito geral (GPIOs), conversores analógico-digitais (ADC) e digital-analógicos (DAC), além de interfaces de comunicação como UART, SPI e I2C.

2.2 Protocolo MQTT

O *Message Queuing Telemetry Transport* (MQTT) é um protocolo de comunicação máquina-para-máquina (M2M) projetado para a Internet das Coisas (IoT). Criado na década de 1990 pela *IBM* e *Eurotech*, ele se tornou um elemento-chave para a comunicação eficiente entre os diversos componentes da indústria, sendo considerado um protocolo padrão para a troca de mensagens nesse meio. Sua concepção foi voltada para conectar dispositivos remotos com baixo consumo de rede e banda, sendo ideal para ambientes com recursos limitados de software e *hardware*. Graças a essas características, o MQTT tem sido amplamente utilizado em setores como automotivo, manufatura, telecomunicações, petróleo e gás.

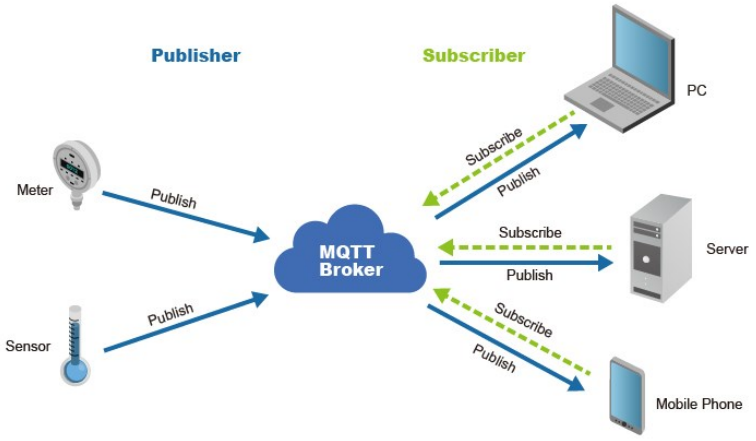
A arquitetura do protocolo é baseada no modelo TCP/IP e utiliza um sistema de mensagens do tipo publicação/subscrição (*publisher/subscriber*). Neste modelo, a comunicação não ocorre diretamente entre os clientes, mas é intermediada por um servidor central chamado *Broker*. O *Broker* é responsável por receber as mensagens publicadas pelos clientes e encaminhá-las a todos os outros clientes que se subscreveram nos tópicos correspondentes. Isso permite que um dispositivo atue simultaneamente como publicador e assinante, facilitando a troca contínua e bidirecional de informações. Essa estrutura descentralizada é uma das grandes vantagens do MQTT, pois a entrada e saída de novos componentes na rede ocorre de forma simples, sem a necessidade de reconfigurar todo o sistema.

A comunicação é organizada por meio de "tópicos", que funcionam como canais para onde as mensagens são enviadas. Um cliente publicador envia uma mensagem (chamada de *payload*) para um tópico específico, e os clientes subscritores (*subscribers*) que se inscreveram nesse tópico

recebem essa mensagem. Esse modelo permite que cada componente da rede tenha autonomia para definir quais informações ele pode transmitir e receber. Para otimizar o uso da banda de rede, os cabeçalhos das mensagens MQTT são projetados para serem pequenos, variando de dois a cinco bytes, com um tamanho máximo de pacote de 256 MB.

O protocolo também oferece três níveis de Qualidade de Serviço (QoS) para garantir a entrega das mensagens. O QoS 0 (no máximo uma vez) garante a entrega da mensagem apenas uma vez, sem a necessidade de confirmação de recebimento (ACK). O QoS 1 (pelo menos uma vez) assegura que a mensagem seja entregue ao menos uma vez, exigindo uma confirmação (PUBACK) do *Broker*. Por fim, o QoS 2 (exatamente uma vez) é o nível mais confiável, garantindo que a mensagem seja entregue exatamente uma vez através de um processo de *handshake* mais complexo. A segurança na comunicação pode ser implementada utilizando o protocolo TLS (*Transport Layer Security*). Vemos na Figura 2 um esquema dessa comunicação. (Maschietto, 2021)

Figura 2 - Comunição MQTT



Fonte: https://oringnet.com/en/knowledge-base/what-is-mqtt-and-how-it-works

2.3 *Flutter*, *Dart* e *Visual Studio Code*

O *Visual Studio Code (VS Code*), desenvolvido pela Microsoft, consolidou-se como um dos editores de código-fonte mais populares e versáteis da atualidade. Lançado em 2015, ele se diferencia de Ambientes de Desenvolvimento Integrado (IDEs) mais pesados por sua natureza leve, rápida e multiplataforma, com suporte nativo para Windows, macOS e Linux.

A filosofia do *VS Code* é fornecer uma base sólida e performática para edição de código, enriquecida por um vasto "ecossistema" de extensões que permitem adaptá-lo para praticamente qualquer linguagem de programação, *framework* ou tecnologia. (Rissi, 2025)

Seus recursos nativos incluem depuração integrada, controle de versionamento *Git* incorporado, realce de sintaxe avançado, e o *IntelliSense*, um poderoso sistema de autocompletar código que oferece sugestões inteligentes baseadas em definições de tipo, módulos e funções. É justamente essa arquitetura baseada em extensões que transforma o *VS Code* de um simples editor de texto em um ambiente de desenvolvimento robusto e personalizado para tecnologias específicas, como é o caso de *Dart* e *Flutter*.

O *Flutter* é um kit de ferramentas de interface de usuário (UI *toolkit*) de código aberto criado pelo Google, projetado para construir aplicações compiladas nativamente para mobile (iOS e Android), web, desktop e sistemas embarcados a partir de uma única base de código. Sua principal proposta de valor é permitir que desenvolvedores criem interfaces ricas, fluidas e performáticas de maneira rápida e consistente em todas as plataformas.

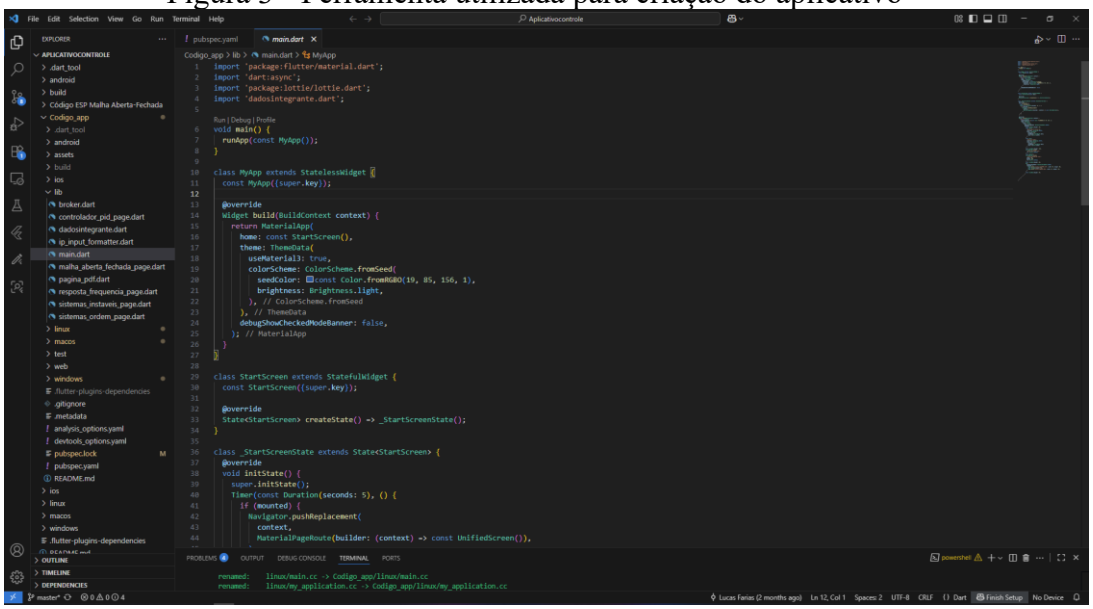
Toda a lógica e a interface de um aplicativo *Flutter* são escritas na linguagem de programação *Dart*, também desenvolvida pelo Google. A *Dart* foi otimizada para o desenvolvimento de aplicações cliente, focando em produtividade e performance. Uma de suas características mais importantes é o suporte a compilação *Just-in-Time* (JIT), que possibilita o recurso de *Hot Reload* durante o desenvolvimento, e *Ahead-of-Time* (AOT), que compila o código para linguagem de máquina nativa, garantindo um desempenho excepcional em produção.

Embora o *Flutter* seja o principal caso de uso para a linguagem *Dart* (*frontend*), ela também é totalmente capaz de ser executada no servidor (*backend*).

A popularidade do *Visual Studio Code* no "ecossistema" *Flutter* não é acidental. A integração entre o editor e as tecnologias do Google é mantida oficialmente e considerada de primeira classe. (Flutter um framework para desenvolvimento mobile, 2025)

Na figura 3 é apresentado o visual do *VS Code* com parte do código dessa solução.

Figura 3 - Ferramenta utilizada para criação do aplicativo

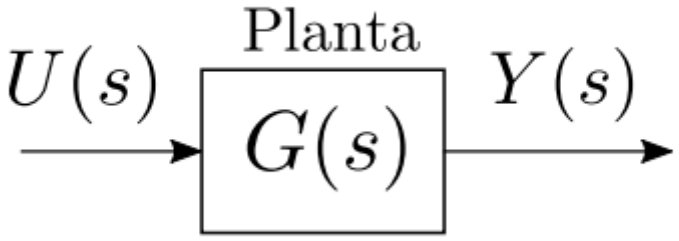


2.4 Controle Linear (Malha Aberta x Malha Fechada)

A engenharia de controle tem como objetivo fundamental o conhecimento e o controle de sistemas para benefício da humanidade. Controlar um sistema pode ser definido como o ato de impor um comportamento desejado a um determinado processo, o que tipicamente envolve guiar a saída desse sistema para um valor de referência com requisitos específicos de velocidade e precisão. Para atingir esse objetivo, duas estruturas de controle são primordiais: o controle em malha aberta (MA) e o controle em malha fechada (MF).

O controle em malha aberta é uma estratégia na qual a entrada da planta é manipulada diretamente para se obter a saída desejada. Nessa configuração, não há um mecanismo de realimentação; a ação de controle é pré-determinada e não leva em conta o resultado real da saída do sistema. Para que funcione adequadamente, é necessário que a relação entre entrada e saída seja conhecida com precisão, pois o sistema é altamente vulnerável a descasamentos de modelo e perturbações externas. Caso existam distúrbios, o sistema continuará operando de acordo com a referência de entrada pré-estabelecida, o que pode ocasionar erros significativos. Na Figura 4, a seguir temos o diagrama de blocos do controle em MA, conforme apresentado durante a aula prática da disciplina de Controle Linear no curso de Engenharia Mecatrônica da Universidade Federal de Uberlândia ao qual temos o seu roteiro apresentado no Anexo A.

Figura 4 - Estrutura de controle em malha aberta

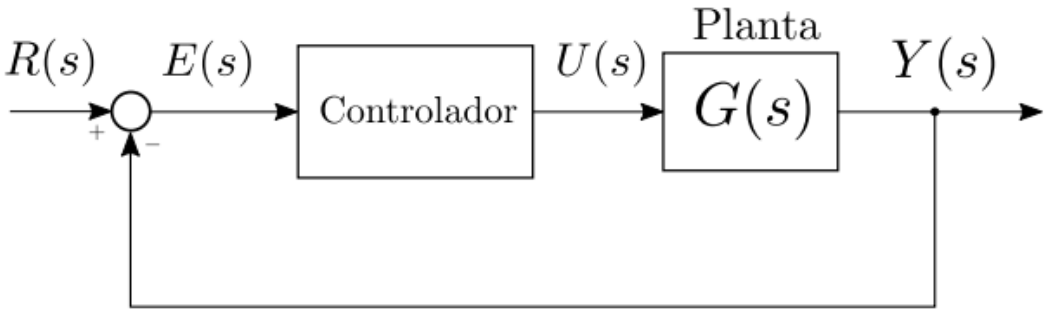


Fonte: Relatório de Laboratório 2 - Malha Aberta *vs* Malha Fechada Prof. Pedro Augusto

As principais vantagens dessa estrutura são a simplicidade de construção e manutenção, o baixo custo de implementação e a conveniência em situações nas quais a medição da saída do processo não é viável ou é economicamente impraticável. Em contrapartida, sua maior desvantagem é a falta de robustez, podendo necessitar de recalibração frequente para manter um desempenho adequado.

Por outro lado, o controle em malha fechada, um fundamento básico da teoria de controle, utiliza o princípio da realimentação (*feedback*). Nessa estrutura, as informações da saída do sistema são medidas e comparadas com um valor de referência desejado. A diferença entre a referência e a saída, conhecida como erro de rastreamento, é utilizada pelo controlador para calcular a entrada adequada para a planta. A principal diferença, portanto, é que o sistema em malha fechada possui realimentação, fazendo com que os valores de saída interfiram diretamente nos valores de entrada, como vemos representado na Figura 5, com o diagrama de blocos para MF. (Pedro Augusto, 2022)

Figura 5 - Estrutura de controle em malha fechada.



Fonte: Relatório de Laboratório 2 - Malha Aberta *vs* Malha Fechada Prof. Pedro Augusto

No controle em malha fechada, a ação de controle é baseada no erro entre o valor desejado (referência) e o valor medido da saída. O componente responsável por calcular essa ação é o

controlador. Existem diversas estratégias para projetar um controlador, variando em complexidade e aplicação. Para os experimentos deste trabalho, foram implementadas e estudadas as seguintes abordagens, com base nos conceitos da teoria de controle linear:

- Controlador Proporcional (P)

O controlador proporcional é a forma mais simples de controle por realimentação. Sua função de transferência é representada apenas por um ganho constante, K. A ação de controle é diretamente proporcional ao sinal de erro. Matematicamente, a saída do controlador é o erro multiplicado pelo ganho K. Embora seja de simples implementação, este tipo de controlador geralmente não consegue zerar o erro em regime permanente para certas entradas, como a rampa, mas pode ser suficiente para estabilizar sistemas que são instáveis em malha aberta.

- Controlador Proporcional-Integral (PI)

Para eliminar o erro em regime permanente, que é uma limitação do controlador puramente proporcional, adiciona-se uma ação integral. O controlador Proporcional-Integral (PI) combina a ação proporcional com uma ação que é proporcional à integral do erro ao longo do tempo. A parcela integral tem a capacidade de acumular o erro passado, garantindo que, mesmo para erros pequenos, a ação de controle continue a aumentar até que o erro seja completamente zerado. Esta característica é especialmente útil em sistemas que precisam seguir uma referência constante sem desvios permanentes.

- Controlador Proporcional-Integral-Derivativo (PID)

O controlador PID é um dos mais utilizados na indústria e adiciona uma terceira componente à lógica de controle: a ação derivativa. Esta ação é proporcional à taxa de variação do erro, ou seja, à sua derivada. A principal função do termo derivativo é antecipar o comportamento futuro do erro, promovendo uma ação corretiva que amortece as oscilações e melhora a resposta transitória do sistema, tornando-o mais estável e rápido. A função de transferência de um controlador PID combina as três ações: proporcional, integral e derivativa.

- Controladores de Avanço e Atraso de Fase (*Lead-Lag*)

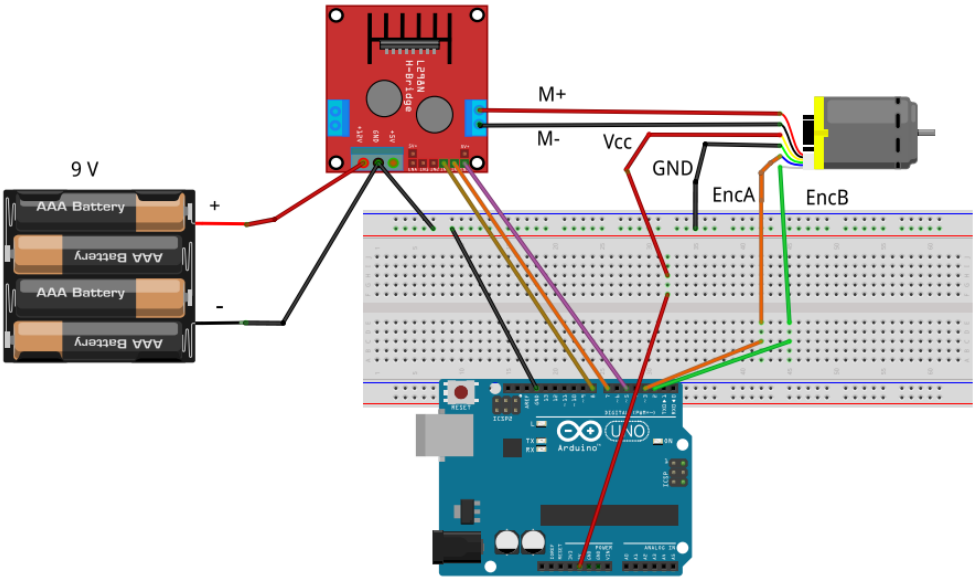
Os controladores de avanço (*Lead*) e atraso (*Lag*) são projetados para alterar a resposta do sistema em regiões específicas. Um controlador de avanço de fase é utilizado para melhorar a resposta transitória do sistema, como o tempo de subida e o *overshoot*. Ele consegue isso

adicionando fase positiva ao sistema, o que melhora a margem de estabilidade. Por outro lado, um controlador de atraso de fase é projetado para melhorar a resposta em regime permanente, reduzindo o erro estacionário, mas com pouca influência sobre o transitório. Ambas as técnicas são implementadas através de circuitos com amplificadores operacionais ou digitalmente, e seu projeto envolve o posicionamento estratégico de polos e zeros para moldar a resposta do sistema conforme as especificações desejadas.

A grande vantagem do controle em MF é a sua robustez a perturbações externas e a descasamentos de modelo, permitindo maior precisão e estabilidade ao processo. Contudo, essa estrutura é inerentemente mais complexa, demandando uma maior quantidade de componentes (como sensores para medir a saída), o que eleva seu custo de implementação e manutenção em comparação com a malha aberta.

A aplicação prática para comparar essas duas estratégias, como realizado durante a aplicação das aulas práticas de Controle Linear na Universidade Federal de Uberlândia, no Laboratório de Ensino da Mecatrônica (LEM3), frequentemente envolve o controle de velocidade de um motor de corrente contínua (CC), como o demonstrado na Figura 6.

Figura 6 - Montagem do modelo utilizado nas aulas de Controle Linear



Fonte: Relatório de Laboratório 2 - Malha Aberta *vs* Malha Fechada Prof. Pedro Augusto

Em malha aberta, varia-se o ciclo ativo de uma onda PWM (a entrada) para se atingir uma velocidade desejada, mas fatores não modelados, como o atrito seco, fazem com que o motor não parta com um *duty cycle* muito baixo e geram imprecisão na relação entre entrada e saída.

Já em malha fechada, um controlador ajusta continuamente o *duty cycle* com base na diferença entre a velocidade de referência e a velocidade real medida por um *encoder*, corrigindo os desvios e garantindo que a saída convirja para o valor desejado com maior precisão. Essa

abordagem ilustra como a realimentação pode estabilizar sistemas e zerar o erro em regime permanente, um objetivo crucial em inúmeras aplicações de controle. (Teixeira, 2013)

3 METODOLOGIA

Durante essa seção serão apresentados os materiais utilizados na estruturação da arquitetura do projeto, as etapas do desenvolvimento dos códigos do aplicativo, bem como os códigos utilizados nos microcontroladores ESP32.

3.1 Arquitetura do Projeto

Baseado no protocolo de comunicação MQTT para conexão entre os dispositivos utilizados, o projeto segue o modelo cliente-servidor, publicando mensagens no servidor (*Publisher*) ou recebendo as mensagens publicadas pelo servidor (*Subscriber*). Para o manuseio do protótipo utilizado no laboratório teremos o aplicativo *mobile*, instalado em um celular com sistema operacional *Android*, no qual teremos a possibilidade de configuração do *Broker*, acesso com identificação de usuários que realizarão os experimentos, escolha do módulo da disciplina de Controle Linear utilizado no experimento, input dos parâmetros de controle e acompanhamento dos dados coletados durante o experimento. Abaixo destrincharemos os principais componentes utilizados no sistema de comunicação:

- **Aplicativo para dispositivo móvel (*Celular Android*)**

O aplicativo atua em duas frentes como cliente: enviando os tópicos para o *broker* (*Publisher*), que atuam como os comandos para o protótipo que se encontra no laboratório. E como *Subscriber* que recebe os dados captados pelo *encoder* atualizados do servidor.

- **Servidor (*Broker* MQTT)**

O *broker* local, em servidor (computador) localizado no laboratório do MAPL, centraliza todas as informações publicadas pelos clientes durante a realização do experimento, tanto do aplicativo como dos microcontroladores ESP32, cada um identificado pela inscrição nos seus respectivos tópicos, garantido assim a comunicação em duas vias (recebendo os dados do ESP32 e enviando para o celular concomitantemente) com adaptabilidade e confiabilidade na troca de dados.

- **Microcontroladores (ESP32)**

O Esp32 é utilizado como executor da lei de controle implementada pelo usuário no aplicativo, fazendo com que as ações sejam refletidas no protótipo físico.

3.2 Materiais e Softwares utilizados

A seguir será apresentado a lista de materiais e softwares utilizados para o desenvolvimento do projeto.

- 1 Microcontrolador ESP32;
- 1 Ponte H – L298N;
- 1 Fonte DC;
- 1 Motor DC;
- 1 Encoder;
- 1 Notebook;
- 1 Broker instalado no computador;
- 1 Celular com sistema operacional Android;
- Fios de conexão;
- Software IDE Arduino;
- Software Android Studio;
- Software *Visual Studio Code*.

3.3 Código Microcontrolador ESP32

Conforme falado anteriormente utilizaremos um microcontrolador ESP32, onde teremos implementado toda a lógica de controle.

É importante ressaltar que a arquitetura proposta neste trabalho, baseada no microcontrolador ESP32 e no controle via aplicativo, difere significativamente da montagem tradicional detalhada no roteiro de laboratório da disciplina. O procedimento experimental aqui descrito, que envolve a configuração do hardware, a conexão com o *broker* MQTT e a utilização da interface do aplicativo móvel, representa uma modernização da prática. Portanto, para a plena adoção desta solução na disciplina, seria necessária uma atualização do material didático fornecido aos alunos, alinhando as instruções à nova arquitetura de controle desenvolvida.

3.3.1 Estrutura do código e Bibliotecas utilizadas

O desenvolvimento do *firmware* inicia-se com a inclusão das bibliotecas essenciais para o funcionamento do projeto, como mostrado na Figura 7.

A biblioteca *Arduino.h* serve como base para todo o *framework*, enquanto *WiFi.h* oferece as funcionalidades para a conexão do ESP32 à rede sem fio. A comunicação com o *broker* MQTT é implementada pela biblioteca *PubSubClient.h*, que permite ao dispositivo publicar e subscrever em tópicos. Por fim, a biblioteca *Encoder.h*, desenvolvida por Paul Stoffregen, em 2009, foi utilizada para a leitura precisa dos pulsos do *encoder* acoplado ao motor, permitindo o cálculo da velocidade de rotação.

O *encoder* utilizado no projeto é do tipo incremental de quadratura. Seu funcionamento baseia-se em um disco com fendas acoplado ao eixo do motor, que, ao girar, gera uma sequência de pulsos elétricos em dois canais (A e B). Ao contar esses pulsos em um intervalo de tempo, o microcontrolador consegue determinar não apenas o deslocamento angular, mas também a direção do giro, possibilitando o cálculo preciso da velocidade.

Figura 7 - Implementação de bibliotecas

```
1   // Includes de bibliotecas
2   #include <Arduino.h>
3   #include <WiFi.h>
4   #include <PubSubClient.h>
5   #include <Encoder.h>
6
```

3.3.2 Configuração e mapeamento de hardware

Para garantir a flexibilidade e a fácil configuração do sistema, as credenciais de rede (SSID e senha) e os dados do servidor MQTT (endereço do *broker* e porta) foram definidos como constantes globais. Essa abordagem permite que o sistema seja rapidamente adaptado para diferentes redes e brokers sem a necessidade de alterar a lógica principal do código, demonstrado na Figura 8.

A interface entre o software e o hardware foi estabelecida através do mapeamento dos pinos do ESP32. Foram designados os pinos de saída para controlar a ponte H L298N, que é responsável por acionar o motor CC, utilizando um pino para o sinal PWM de velocidade (ENB) e dois pinos para o sentido de rotação (IN3 e IN4). Adicionalmente, os pinos de entrada foram definidos para receber os sinais dos canais A e B do *encoder* do motor, que são fundamentais para a medição da velocidade em malha fechada.

Figura 8 - Configuração de conexão e hardware

```
7   // ============================================================
8   // == 1. CONFIGURAÇÕES
9   // ============================================================
10  const char* SSID = "NOME_REDE";
11  const char* PASSWORD = "SENHA_REDE";
12  const char* MQTT_BROKER = "IP_BROKER";
13  const int MQTT_PORT = 1883;
14
15  // ============================================================
16  // == 2. MAPEAMENTO DE PINOS (MOTOR DC COM L298N)
17  // ============================================================
18  // Pinos da Ponte H L298N
19  const int MOTOR_PWM_PIN = 25; // Deve estar ligado ao pino ENB do L298N
20  const int MOTOR_IN3_PIN = 26; // Deve estar ligado ao pino IN3 do L298N
21  const int MOTOR_IN4_PIN = 27; // Deve estar ligado ao pino IN4 do L298N
22
23  // Pinos do Encoder do motor
24  const int ENCODER_A_PIN = 34;
25  const int ENCODER_B_PIN = 35;
```

3.3.3 Variáveis de Estado e Controle

A lógica de controle é gerenciada por um conjunto de variáveis globais. Foi instanciado um objeto da classe *Encoder* para gerenciar a leitura do sensor de rotação. Para organizar a operação do sistema, foi definido um enumerador *ControlMode* com três estados possíveis: PARADO, MALHA_ABERTA e MALHA_FECHADA. A variável *currentMode* armazena o estado atual, determinando qual lógica de controle será executada.

Foram criadas variáveis para armazenar os parâmetros recebidos via MQTT, como o sinal de controle em malha aberta (*param_u_MA*), a referência de velocidade em malha fechada (*param_ref_MF*) e o ganho proporcional (*param_Kp_MF*). Outras variáveis globais armazenam dados em tempo real, como o sinal de controle efetivamente aplicado (*u_control*), o erro de rastreamento (*erro_MF*) e a velocidade calculada (*vel_rpm*), que são utilizados tanto nos cálculos de controle quanto para publicação via MQTT, evidenciados na Figura 9.

Figura 9 - Declaração de variáveis

```
// ============================================================
// == 4. OBJETOS E VARIÁVEIS DE ESTADO
// ============================================================
Encoder encoder(ENCODER_A_PIN, ENCODER_B_PIN);

enum ControlMode { PARADO, MALHA_ABERTA, MALHA_FECHADA };
ControlMode currentMode = PARADO;

float param_u_MA = 0.0;
float param_ref_MF = 0.0;
float param_Kp_MF = 0.1;

float u_control = 0.0;
float erro_MF = 0.0;
float vel_rpm = 0.0;

long encoder_pos = 0;
long encoder_pos_ant = 0;

unsigned long time_curr = 0, time_prev = 0;
double dt_sec = 0.0;

unsigned long lastControlTime = 0;
unsigned long lastMqttPublishTime = 0;
```

3.3.4 Inicialização do Sistema

Na Figura 10, é mostrado a função *setup*() é executada uma única vez quando o ESP32 é inicializado e é responsável por configurar todos os componentes de hardware e software. Primeiramente, a comunicação serial é iniciada para fins de depuração. Em seguida, os pinos de controle do motor são configurados como saídas, e um sentido de rotação padrão é estabelecido. A conexão com a rede Wi-Fi é estabelecida através da chamada da função *setup_wifi(),* e, subsequentemente, o cliente MQTT é configurado com o endereço do broker e a função *callback*, que tratará as mensagens recebidas.

Figura 10 - Função Setup

```
71  // ============================================================
72  // == 7. FUNÇÃO SETUP
73  // ============================================================
74  void setup() {
75    Serial.begin(115200);
76    Serial.println("\n>>> Iniciando ESP32 - Controle de Motor via MQTT <<<");
77
78    // --- Configuração dos Pinos do Motor -
79    pinMode(MOTOR_IN3_PIN, OUTPUT);
80    pinMode(MOTOR_IN4_PIN, OUTPUT);
81    pinMode(MOTOR_PWM_PIN, OUTPUT);
82
83    // Define um sentido de giro padrão usando IN3 e IN4
84    digitalWrite(MOTOR_IN3_PIN, HIGH);
85    digitalWrite(MOTOR_IN4_PIN, LOW);
86
87    analogWrite(MOTOR_PWM_PIN, 0); // Controla a velocidade via ENB
88    Serial.println("-> Pinos do Motor configurados para o Canal B (ENB, IN3, IN4).");
89
90    Serial.println("-> Encoder configurado para a biblioteca de Paul Stoffregen.");
91
92    setup_wifi();
93    client.setServer(MQTT_BROKER, MQTT_PORT);
94    client.setCallback(callback);
95
96    time_prev = micros();
97  }
98
```

3.3.5 Lógica de Controle e Comunicação

A função *loop*() constitui o coração do *firmware*, sendo executada continuamente. Para garantir um comportamento previsível e não bloqueante, a lógica foi dividida em tarefas temporizadas utilizando a função *millis*().

A principal tarefa é o ciclo de controle, que é executado a cada 20 milissegundos. Dentro deste ciclo, a primeira etapa é a leitura da posição atual do *encoder*. Com base na posição atual, na posição anterior e no intervalo de tempo decorrido (*dt_sec*), a velocidade do motor em RPM (*vel_rpm*) é calculada.

Com a velocidade atualizada, a lógica de controle é acionada através de uma estrutura *switch*, que seleciona a ação a ser tomada com base na variável *currentMode*:

- Malha Aberta: O sinal de controle *u_control* recebe diretamente o valor do parâmetro *param_u_MA*, enviado pelo aplicativo.

- Malha Fechada: O erro (*erro_MF*) é calculado como a diferença entre a referência (*param_ref_MF*) e a velocidade medida (*vel_rpm*). A lei de controle proporcional

é então aplicada, definindo *u_control* como o produto do erro pelo ganho *param_Kp_MF*.

- Parado: O sinal de controle é definido como zero para manter o motor inerte.

Após o cálculo, o sinal de controle *u_control* é saturado entre 0 e 90% para garantir a segurança da operação e, em seguida, é convertido para a escala de 8 *bits* (0-255) do PWM e aplicado ao motor através da função *analogWrite*().

Uma segunda tarefa temporizada, executada a cada 500 milissegundos, é responsável por publicar os dados do sistema (velocidade, modo de controle, erro etc.) no *broker* MQTT, permitindo que o aplicativo móvel monitore o experimento em tempo real, assim como detalhado na Figura 11.

Figura 11 - Detalhamento programação da lógica de controle

```
102  void loop() {
103    if (!client.connected()) {
104      reconnect();
105    }
106    client.loop();
107
108    unsigned long now = millis();
109
110    if (now - lastControlTime >= 20) {
111      lastControlTime = now;
112
113      encoder_pos_ant = encoder_pos;
114      encoder_pos = encoder.read();
115
116      double pos_deg = (encoder_pos * 360.0) / ENCODER_PPR;
117      double pos_deg_ant = (encoder_pos_ant * 360.0) / ENCODER_PPR;
118
119      time_curr = micros();
120      dt_sec = (time_curr - time_prev) / 1000000.0;
121      time_prev = time_curr;
122
123      if (dt_sec > 0) {
124        vel_rpm = ((pos_deg - pos_deg_ant) / dt_sec) / 6.0;
125      }
126
127      switch (currentMode) {
128        case MALHA_ABERTA:
129          u_control = param_u_MA;
130          erro_MF = 0;
131          break;
132        case MALHA_FECHADA:
133          erro_MF = param_ref_MF - vel_rpm;
134          u_control = param_Kp_MF * erro_MF;
135          break;
136        case PARADO:
137        default:
138          u_control = 0;
139          erro_MF = 0;
140          break;
141      }
142
143      u_control = constrain(u_control, 0.0, 90.0);
144      analogWrite(MOTOR_PWM_PIN, map(u_control, 0, 100, 0, 255));
145    }
146
147    if (now - lastMqttPublishTime >= 500) {
148      lastMqttPublishTime = now;
149      if (client.connected()) {
150        publishMqttData();
```

3.3.6 Funções de Comunicação MQTT

A comunicação remota é gerenciada por um conjunto de funções auxiliares, evidenciadas na Figura 12. A função *callback*() é fundamental, pois é executada automaticamente sempre que o ESP32 recebe uma mensagem em um dos tópicos subscritos. Ela interpreta o tópico e o conteúdo da mensagem para atualizar os parâmetros de controle (*uMA*, *refMF*, *KpMF*) ou para executar comandos específicos, como parar o motor. É esta função que permite a troca do modo de operação entre malha aberta e malha fechada com base nos comandos do usuário no aplicativo.

Figura 12 – Funções auxiliares (call-back)

```
158  void setup_wifi() {
159    delay(10);
160    Serial.print("\nConectando a rede Wi-Fi: ");
161    Serial.println(SSID);
162    WiFi.begin(SSID, PASSWORD);
163    while (WiFi.status() != WL_CONNECTED) {
164      delay(500);
165      Serial.print(".");
166    }
167    Serial.println("\nWi-Fi conectado!");
168    Serial.print("Endereço IP: ");
169    Serial.println(WiFi.localIP());
170  }
171
172  void callback(char* topic, byte* payload, unsigned int length) {
173    String message;
174    message.reserve(length);
175    for (unsigned int i = 0; i < length; i++) {
176      message += (char)payload[i];
177    }
178
179    String topicStr = String(topic);
180    Serial.printf("Mensagem recebida -> Tópico: %s, Valor: %s\n", topicStr.c_str(), message.c_str());
181
182    float value = message.toFloat();
183
184    if (topicStr == "uMA") {
185      param_u_MA = value;
186      currentMode = MALHA_ABERTA;
187      Serial.printf("-> MODO: Malha Aberta. U = %.2f %%\n", param_u_MA);
188    } else if (topicStr == "refMF") {
189      param_ref_MF = value;
190      currentMode = MALHA_FECHADA;
191      Serial.printf("-> MODO: Malha Fechada. Referência = %.2f RPM\n", param_ref_MF);
192    } else if (topicStr == "KpMF") {
193      param_Kp_MF = value;
194      Serial.printf("-> Parâmetro atualizado: Kp = %.4f\n", param_Kp_MF);
195    } else if (topicStr == "controle/comando") {
196      if (message == "PARAR") {
197        currentMode = PARADO;
198        encoder.write(0);
199        encoder_pos = 0;
200        vel_rpm = 0;
201        Serial.println("-> COMANDO: Parar motor. Contador do encoder zerado.");
202      }
203    }
204  }
```

A função *reconnect*() garante a robustez da comunicação, tentando reconectar-se ao broker MQTT caso a conexão seja perdida. Ao se reconectar, ela também renova a subscrição a todos os

tópicos de comando. Finalmente, a função *publishMqttData*() organiza e envia os dados de estado do sistema, como a velocidade atual, o modo de operação e o erro de controle, para tópicos específicos, permitindo o monitoramento contínuo pelo aplicativo, estruturado como na Figura 13.

Figura 13 - Funções auxiliares (*reconnect* e *publishMqtt*)

```
206  void reconnect() {
207    while (!client.connected()) {
208      Serial.print("Tentando conectar ao Broker MQTT...");
209      String clientId = "ESP32_Motor_Client_";
210      clientId += String(random(0xffff), HEX);
211
212      if (client.connect(clientId.c_str())) {
213        Serial.println("Conectado!");
214        client.subscribe("uMA");
215        client.subscribe("refMF");
216        client.subscribe("KpMF");
217        client.subscribe("controle/comando");
218        Serial.println("Subscrição aos tópicos realizada.");
219      } else {
220        Serial.print("Falha, rc=");
221        Serial.print(client.state());
222        Serial.println(" | Tentando novamente em 5 segundos");
223        delay(5000);
224      }
225    }
226  }
227
228  void publishMqttData() {
229    char msgBuffer[10];
230
231    dtostrf(vel_rpm, 4, 2, msgBuffer);
232    client.publish("motor/velocidade", msgBuffer);
233
234    if(currentMode == PARADO) client.publish("motor/status", "PARADO");
235    else if(currentMode == MALHA_ABERTA) client.publish("motor/status", "MALHA_ABERTA");
236    else if(currentMode == MALHA_FECHADA) client.publish("motor/status", "MALHA_FECHADA");
237
238    if (currentMode == MALHA_FECHADA) {
239      dtostrf(u_control, 4, 2, msgBuffer);
240      client.publish("uMF", msgBuffer);
241      dtostrf(erro_MF, 4, 2, msgBuffer);
242      client.publish("erroMF", msgBuffer);
243    }
244  }
```

3.4 Código Aplicativo

O desenvolvimento do aplicativo móvel, que serve como Interface Homem-Máquina (IHM) para o controle remoto dos experimentos, foi realizado no ambiente do *Visual Studio Code*. A escolha tecnológica recaiu sobre o *framework Flutter* e a linguagem de programação *Dart*, ambos desenvolvidos pelo Google. Essa decisão foi motivada pela capacidade do *Flutter* de criar aplicações multiplataforma compiladas nativamente a partir de uma única base de código,

garantindo uma experiência de usuário fluida e consistente tanto em dispositivos Android quanto iOS.

A metodologia de desenvolvimento foi focada em criar uma aplicação modular, intuitiva e robusta, com funcionalidades claras de navegação, entrada de dados, comunicação e suporte teórico.

3.4.1 Arquitetura e telas iniciais

O aplicativo inicia com uma tela de apresentação, conforme ilustrado na Figura 14, que, após um breve período, direciona o usuário para a interface principal de conexão.

Figura 14 - Tela de inicialização

A estrutura do projeto é modular, com cada funcionalidade principal encapsulada em seu próprio arquivo *.dart*, como *dadosintegrante.dart* para a conexão e *broker.dart* para a seleção de experimentos. Foram utilizadas bibliotecas essenciais como *lottie* para animações, *connectivity_plus* para verificação de rede, *mqtt_client* para a comunicação e *pdfx* para a visualização de documentos, conforme mostrado na Figura 15.

Figura 15 - Parte do código criado (*main.dart*)

```dart
import 'package:flutter/material.dart';
import 'dart:async';
import 'package:lottie/lottie.dart';
import 'dadosintegrante.dart';

Run | Debug | Profile
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

```

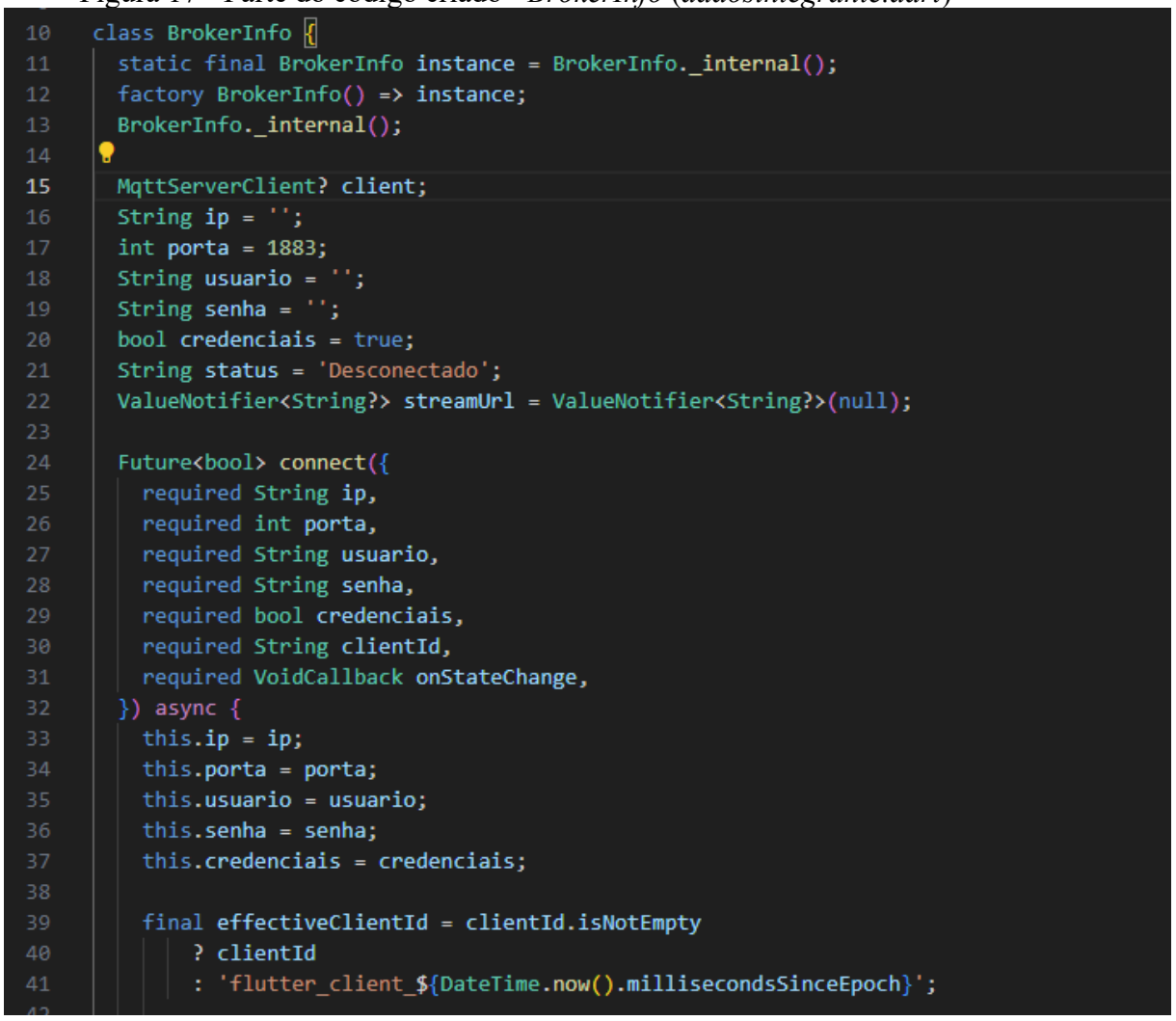
A primeira tela interativa é a de "Dados e Conexão", apresentada na Figura 16. Nesta interface, o aluno deve fornecer o endereço IP do *broker* MQTT do laboratório e, opcionalmente, suas credenciais de acesso. Além disso, é solicitado o nome completo do aluno, que é utilizado para gerar um *ClientId* único para a sessão MQTT, um requisito fundamental para a identificação do cliente na rede.

Figura 16 - Tela de dados de conexão e dados do aluno

A lógica de conexão é centralizada na classe *BrokerInfo*, implementada como um *Singleton* para garantir uma única instância do cliente MQTT em toda a aplicação, mostrado na Figura 17. A função *_connectOrDisconnect*, evidenciado na Figura 18, presente na tela *UnifiedScreen*,

gerencia o processo, valida os campos de entrada, verifica a conectividade de rede do dispositivo e, em caso de sucesso, estabelece a conexão com o *broker*, fornecendo *feedback* visual ao usuário, como a animação de "Conectado!" vista na Figura 19 e a caixa de diálogo de sucesso mostrada na Figura 20.

Figura 17 - Parte do código criado - *BrokerInfo* (*dadosintegrante.dart*)



```dart
class BrokerInfo {
  static final BrokerInfo instance = BrokerInfo._internal();
  factory BrokerInfo() => instance;
  BrokerInfo._internal();

  MqttServerClient? client;
  String ip = '';
  int porta = 1883;
  String usuario = '';
  String senha = '';
  bool credenciais = true;
  String status = 'Desconectado';
  ValueNotifier<String?> streamUrl = ValueNotifier<String?>(null);

  Future<bool> connect({
    required String ip,
    required int porta,
    required String usuario,
    required String senha,
    required bool credenciais,
    required String clientId,
    required VoidCallback onStateChange,
  }) async {
    this.ip = ip;
    this.porta = porta;
    this.usuario = usuario;
    this.senha = senha;
    this.credenciais = credenciais;

    final effectiveClientId = clientId.isNotEmpty
        ? clientId
        : 'flutter_client_${DateTime.now().millisecondsSinceEpoch}';
```

Figura 18- Parte do código criado - _ *connectOrDisconnect* (*dadosintegrante.dart*)

```dart
Future<void> _connectOrDisconnect() async {
  if (brokerInfo.client != null &&
      brokerInfo.client!.connectionStatus!.state ==
          MqttConnectionState.connected) {
    await brokerInfo.disconnect(onStateChange: () => setState(() {}));
    _clearAllFields();
    _showInfoDialog("Desconectado", "Você foi desconectado do broker.");
  } else {
    if (nameController.text.trim().isEmpty) {
      _showErrorDialog("Campo Obrigatório",
          "Por favor, preencha o nome do aluno para ser usado como Client ID.");
      return;
    }
    if (ipController.text.trim().isEmpty) {
      _showErrorDialog(
          "Campo Obrigatório", "Por favor, informe o IP do broker.");
      return;
    }

    final connectivity = await Connectivity().checkConnectivity();
    if (connectivity == ConnectivityResult.none && mounted) {
      _showErrorDialog(
          "Sem Internet", "Por favor, verifique sua conexão com a internet.");
      return;
    }

    final String clientId = nameController.text
        .trim()
        .toLowerCase()
        .replaceAll(RegExp(r'[^a-z0-9]'), '_');

    showDialog(
      context: context,
      barrierDismissible: false,
      builder: (context) => const AlertDialog(
        content: Row(
          children: [
            CircularProgressIndicator(),
```

Figura 19 - Animação de conexão com *Broker*

Figura 20 - *Pop up* de validação de conexão

3.4.2 Seleção e navegação entre experimentos

Uma vez conectado, o botão "Avançar para Experimentos" é habilitado, levando o usuário à tela de "Seleção de Experimento", ilustrada na Figura 21. Esta tela serve como um menu principal, listando os roteiros de aulas práticas disponíveis. Cada experimento é representado por um botão, que, ao ser pressionado, aciona a função de navegação *_navigateToExperimento*, mostrado na Figura 22. Esta função direciona o usuário para a interface de controle correspondente.

Figura 21 - Tela de navegação entre os experimentos

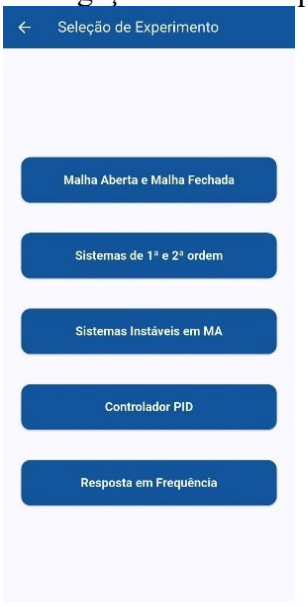
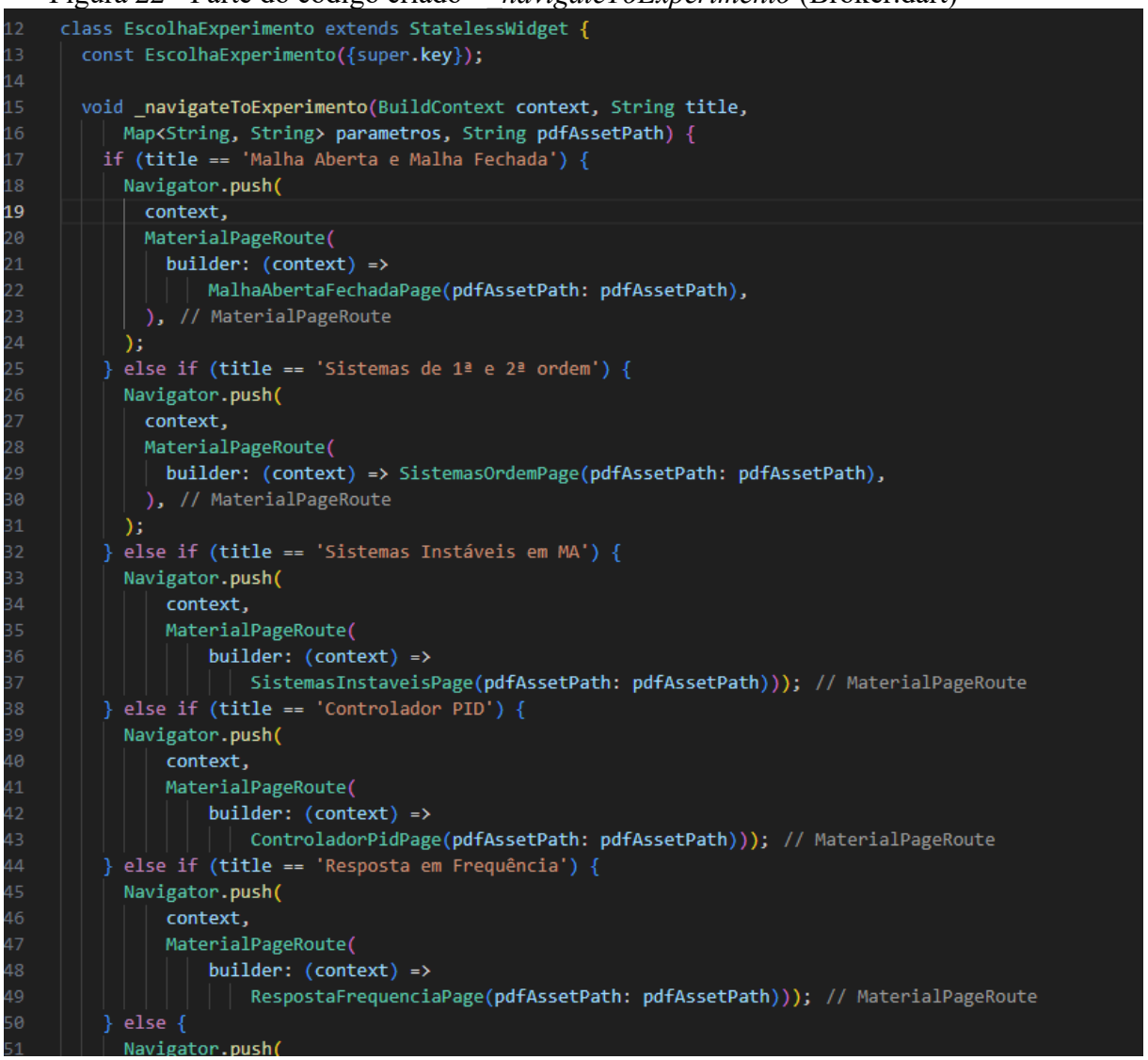


Figura 22 - Parte do código criado - *_navigateToExperimento* (Broker.dart)



```dart
class EscolhaExperimento extends StatelessWidget {
  const EscolhaExperimento({super.key});

  void _navigateToExperimento(BuildContext context, String title,
      Map<String, String> parametros, String pdfAssetPath) {
    if (title == 'Malha Aberta e Malha Fechada') {
      Navigator.push(
        context,
        MaterialPageRoute(
          builder: (context) =>
              MalhaAbertaFechadaPage(pdfAssetPath: pdfAssetPath),
        ), // MaterialPageRoute
      );
    } else if (title == 'Sistemas de 1ª e 2ª ordem') {
      Navigator.push(
        context,
        MaterialPageRoute(
          builder: (context) => SistemasOrdemPage(pdfAssetPath: pdfAssetPath),
        ), // MaterialPageRoute
      );
    } else if (title == 'Sistemas Instáveis em MA') {
      Navigator.push(
          context,
          MaterialPageRoute(
              builder: (context) =>
                  SistemasInstaveisPage(pdfAssetPath: pdfAssetPath))); // MaterialPageRoute
    } else if (title == 'Controlador PID') {
      Navigator.push(
          context,
          MaterialPageRoute(
              builder: (context) =>
                  ControladorPidPage(pdfAssetPath: pdfAssetPath))); // MaterialPageRoute
    } else if (title == 'Resposta em Frequência') {
      Navigator.push(
          context,
          MaterialPageRoute(
              builder: (context) =>
                  RespostaFrequenciaPage(pdfAssetPath: pdfAssetPath))); // MaterialPageRoute
    } else {
      Navigator.push(
```

3.4.3 Interface e monitoramento do experimento

A tela de controle de cada experimento, como a de "Malha Aberta vs. Fechada" apresentada na Figura 23, é o núcleo da interação do aluno com o protótipo físico. A interface é dividida em três áreas principais:

- Controles Gerais: Um botão de "Parar Motor" para interromper a operação a qualquer momento.
- Monitoramento em Tempo Real: Um painel que exibe os dados recebidos do ESP32 via MQTT, como *status*, velocidade, sinal de controle e o erro.
- Entrada de Parâmetros: Seções dedicadas para os modos de Malha Aberta e Malha Fechada.

Figura 23 - Tela interface do experimento

A atualização dos dados de monitoramento é feita de forma reativa. O *widget* *_buildStatusDisplay* utiliza *ValueListenableBuilder* para escutar mudanças nas variáveis de estado, apresentado na Figura 24. Quando uma nova mensagem MQTT chega e atualiza uma variável, a interface gráfica é reconstruída automaticamente para refletir o novo valor.

Figura 24 – Parte do código criado - *_buildStatusDisplay* (malha_aberta_fechada_*page.dart*)

```
222   Widget _buildStatusDisplay() {
223     return Card(
224       elevation: 4,
225       shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(12)),
226       color: Colors.blueGrey[50],
227       child: Padding(
228         padding: const EdgeInsets.all(16.0),
229         child: Column(
230           crossAxisAlignment: CrossAxisAlignment.start,
231           children: [
232             Text(
233               'Monitoramento em Tempo Real',
234               style: Theme.of(context).textTheme.titleLarge?.copyWith(
235                     fontWeight: FontWeight.bold,
236                   ),
237             ), // Text
238             const Divider(height: 20),
239             ValueListenableBuilder<String>(
240               valueListenable: motorStatus,
241               builder: (context, status, child) =>
242                   Text('Status: $status', style: const TextStyle(fontSize: 16)),
243             ), // ValueListenableBuilder
244             const SizedBox(height: 8),
245             ValueListenableBuilder<double>(
246               valueListenable: velocidadeRpm,
247               builder: (context, velocidade, child) => Text(
248                   'Velocidade: ${velocidade.toStringAsFixed(2)} RPM',
249                   style: const TextStyle(fontSize: 16)), // Text
```

3.4.4 Lógica de Comunicação MQTT

A comunicação durante o experimento é bidirecional. A recepção de dados é gerenciada pela função *_setupMqttListener*, demonstrado na Figura 25, que subscreve o aplicativo nos tópicos de telemetria (motor/status, motor/velocidade etc.). Um *StreamSubscription* escuta continuamente por novas mensagens do broker. Ao receber uma mensagem, a função a decodifica e atualiza a variável de estado correspondente, que por sua vez atualiza a UI.

O envio de comandos é feito por funções como _enviarMalhaAberta e _enviarMalhaFechada, mostrado na Figura 26. Elas coletam os dados dos campos de texto, validam a entrada e utilizam a função *_publishMessage* para publicar a mensagem no tópico MQTT apropriado, com um nível de Qualidade de Serviço (QoS) *atLeastOnce* para garantir a entrega.

Figura 25 – Parte do código criado - *setupMqttListener* (malha_aberta_fechada_*page.dart*)

```dart
void _setupMqttListener() {
  if (brokerInfo.client != null &&
      brokerInfo.client!.connectionStatus!.state ==
          MqttConnectionState.connected) {
    const topics = ['motor/status', 'motor/velocidade', 'uMF', 'erroMF'];
    for (var topic in topics) {
      brokerInfo.client!.subscribe(topic, MqttQos.atLeastOnce);
    }

    mqttSubscription = brokerInfo.client!.updates!
        .listen((List<MqttReceivedMessage<MqttMessage?>>? c) {
      if (c != null && c.isNotEmpty) {
        final recMess = c[0].payload as MqttPublishMessage;
        final payload =
            MqttPublishPayload.bytesToStringAsString(recMess.payload.message);
        final topic = c[0].topic;

        switch (topic) {
          case 'motor/status':
            motorStatus.value = payload;
            break;
          case 'motor/velocidade':
            velocidadeRpm.value = double.tryParse(payload) ?? 0.0;
            break;
          case 'uMF':
            uMF.value = double.tryParse(payload) ?? 0.0;
            break;
          case 'erroMF':
            erroMF.value = double.tryParse(payload) ?? 0.0;
            break;
        }
      }
```

Figura 26 - Parte do código criado - *_enviarMalhaFechada* (malha_aberta_fechada_*page.dart*)

```dart
void _enviarMalhaAberta() {
  if (_uMAController.text.isNotEmpty) {
    _publishMessage('uMA', _uMAController.text.replaceAll(',', '.'));
    _showFeedbackDialog('Enviado!', 'Comando de Malha Aberta enviado.');
  } else {
    _showFeedbackDialog('Erro', 'Por favor, insira o valor do Duty Cycle.');
  }
}

void _enviarMalhaFechada() {
  if (_refMFController.text.isNotEmpty && _kpMFController.text.isNotEmpty) {
    _publishMessage('refMF', _refMFController.text.replaceAll(',', '.'));
    _publishMessage('KpMF', _kpMFController.text.replaceAll(',', '.'));
    _showFeedbackDialog('Enviado!', 'Comandos de Malha Fechada enviados.');
  } else {
    _showFeedbackDialog('Erro', 'Preencha a Referência e o Ganho Kp.');
  }
}

void _pararMotor() {
  _publishMessage('controle/comando', 'PARAR');
  _showFeedbackDialog('Enviado!', 'Comando para PARAR o motor foi enviado.');
}
```

3.4.5 Embasamento Teórico

Para auxiliar o aluno durante a prática, um botão de ajuda está presente em todas as telas de experimento. Ao ser acionado, ele abre a tela de "Embasamento Teórico", conforme a Figura 27. Esta funcionalidade, implementada no arquivo *pagina_pdf.dart*, visto na Figura 28 utiliza o pacote *pdfx* para renderizar o roteiro da aula em formato PDF diretamente no aplicativo, permitindo a consulta rápida da teoria.

Figura 27 - Tela de embasamento teórico

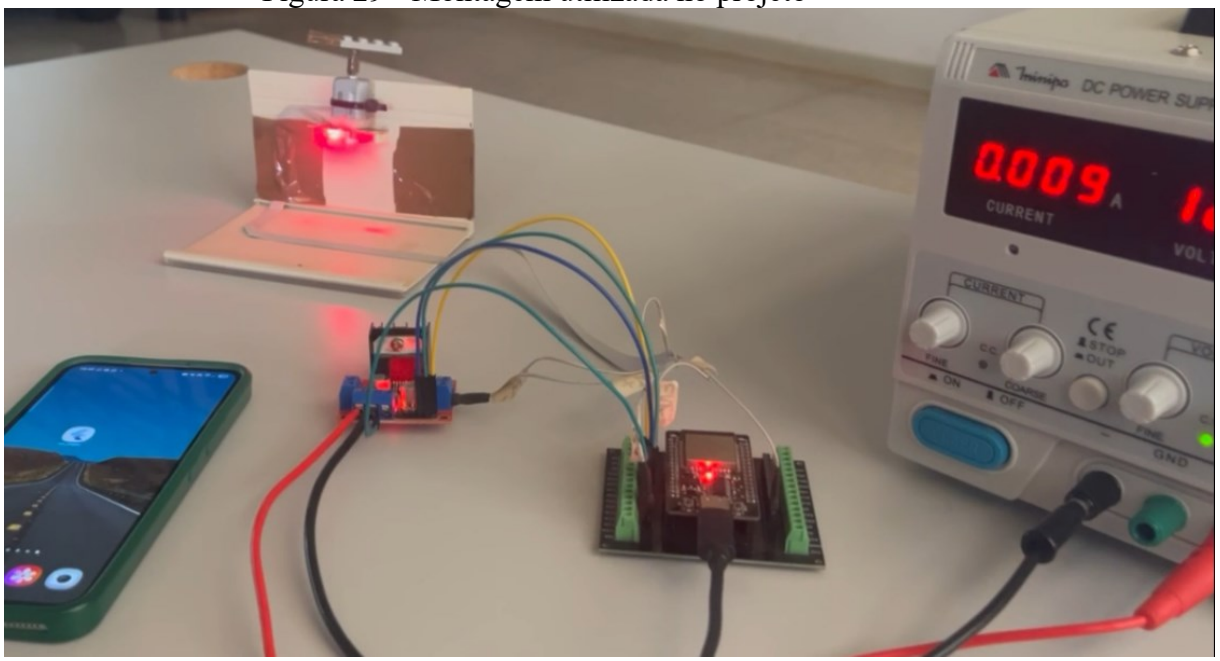
Figura 28 - Parte do código criado - *PdfControllerPinch* (pagina *pdf.dart*)

```dart
import 'package:flutter/material.dart';
import 'package:pdfx/pdfx.dart';

class PdfPage extends StatefulWidget {
  const PdfPage({super.key});

  @override
  State<PdfPage> createState() => _PdfPageState();
}

class _PdfPageState extends State<PdfPage> {
  late PdfControllerPinch pdfControllerPinch;

  int contadorPaginas = 0, paginaAtual = 1;

  @override
  void initState() {
    super.initState();
    pdfControllerPinch = PdfControllerPinch(
        document: PdfDocument.openAsset('assets/pdfs/controlador_pid.pdf'));
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text(
          "Embasamento Teórico",
          style: TextStyle(
            color: Colors.white,
          ), // TextStyle
        ), // Text
        backgroundColor: const Color.fromRGBO(19, 85, 156, 1),
      ), // AppBar
      body: _buildUI(),
    ); // Scaffold
  }

  Widget _buildUI() {
    return Column(
```

4 RESULTADOS

Apresentaremos agora os resultados obtidos na prática com o protótipo do experimento de Malha aberta e Malha fechada utilizados durante a aplicação da disciplina de Controle Linear no LEM 3 demonstrado na Figura 29.

Figura 29 - Montagem utilizada no projeto



Temos a utilização do ESP32 conectado a ponte H – L298N, das quais suas ligações são explicitadas na tabela a seguir.

Tabela 1 - Conexão L298N a ESP32

L298N	ESP32
IN3	GPIO 26
IN4	GPIO 27
ENB	GPIO 25
5V	3V3
GND	GND

Para conexão dos componentes do *encoder* com o ESP32 é utilizado o componente acoplado na mesma placa da ponte H - L298N e é discriminado na Tabela 2, a seguir.

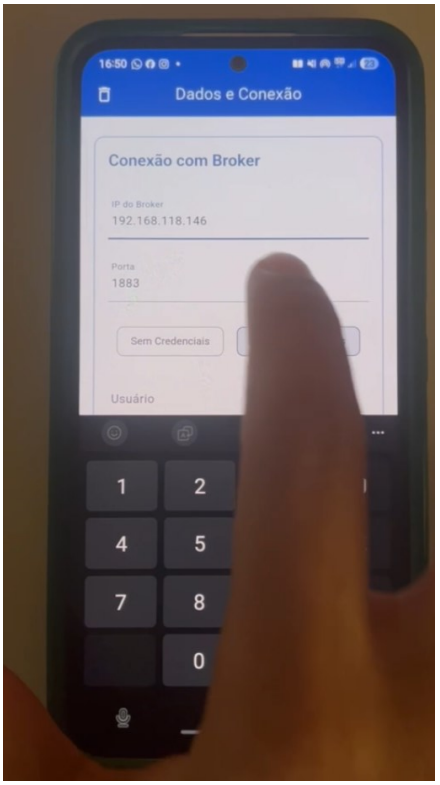
Tabela 2 - Conexão *Encoder* a ESP32

ENCODER	ESP32
ENCA	GPIO 34
ENCB	GPIO 35
VCC	VIN (5V)
GND	GND

A alimentação do motor, como vemos na Figura 29 a alimentação do motor é realizada através do terminal +12V do L298N do qual é conectado no terminal positivo (+) da fonte DC externa. O GND do L298N é conectado no terminal negativo (-).

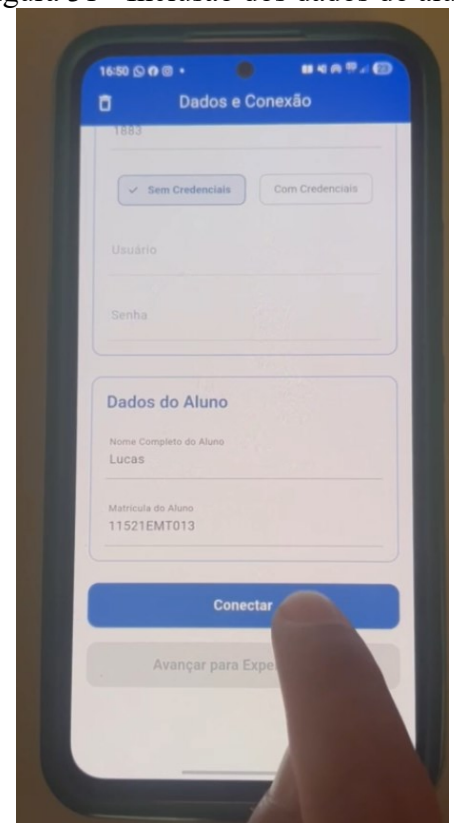
Após realizado as ligações necessárias a fonte é ligada, o aplicativo é inicializado através do celular e os dados do *broker* são inseridos conforme a Figura 30 mostra.

Figura 30 - Inclusão dos dados do broker



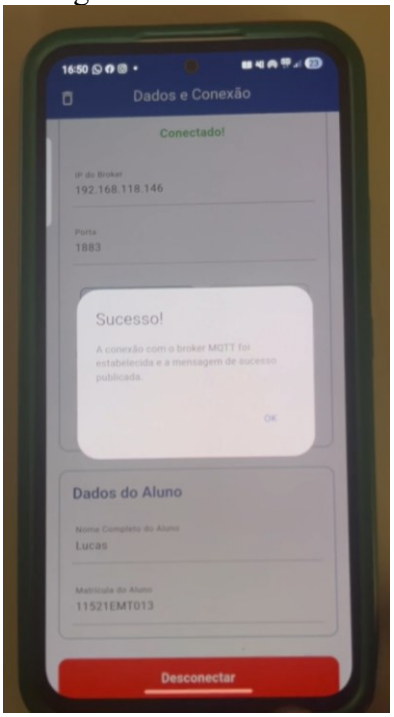
É selecionado que não utilizaremos credenciais e os dados do aluno são inseridos, como mostrado na Figura 31. Neste momento temos a opção de inclusão de mais de um aluno, pois atualmente as aulas são realizadas em grupos de 2 ou mais alunos.

Figura 31 - Inclusão dos dados do aluno



Com os dados corretos e a conexão for estabelecida, temos a mensagem de sucesso na conexão com o *broker*, apresentada na Figura 32.

Figura 32 - Mensagem de status da conexão com broker



Após a conexão com *broker* ser realizada com sucesso, somos direcionados para a tela de escolha do experimento a ser realizado. Como informado anteriormente como demonstração,

iremos focar no experimento Malha Aberta vs. Fechada, ao qual iremos acessar clicando no botão de mesmo nome.

O usuário será direcionado para a tela de interação com o experimento, onde será possível parar o motor remotamente, monitorar os dados em tempo real, incluir os dados de *duty cycle* (%), para o módulo de malha aberta, onde enviamos a informação de 50% para o *broker*, como mostrado na Figura 33 que vemos no painel do MQTT Explorer e no Monitor Serial do ESP32 a velocidade de rotação do motor, o status que mostra que os dados de MALHA_ABERTA está em atividade e o dado do *duty cycle* enviado (uMA), conforme a Figura 34.

Figura 33 - Interface com experimento

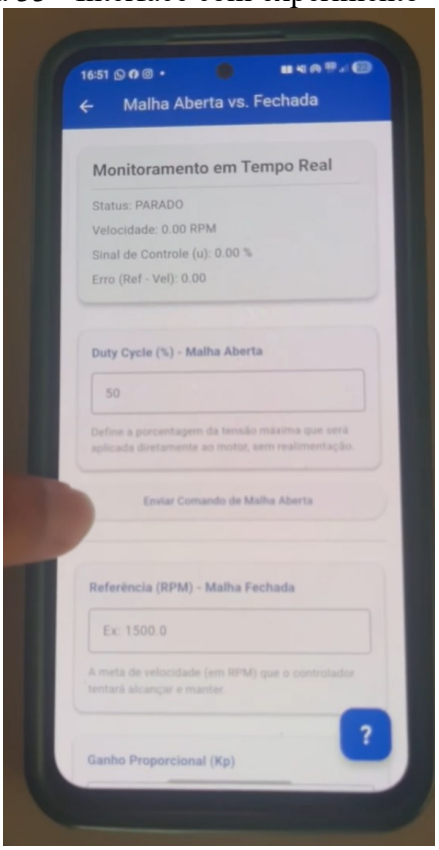
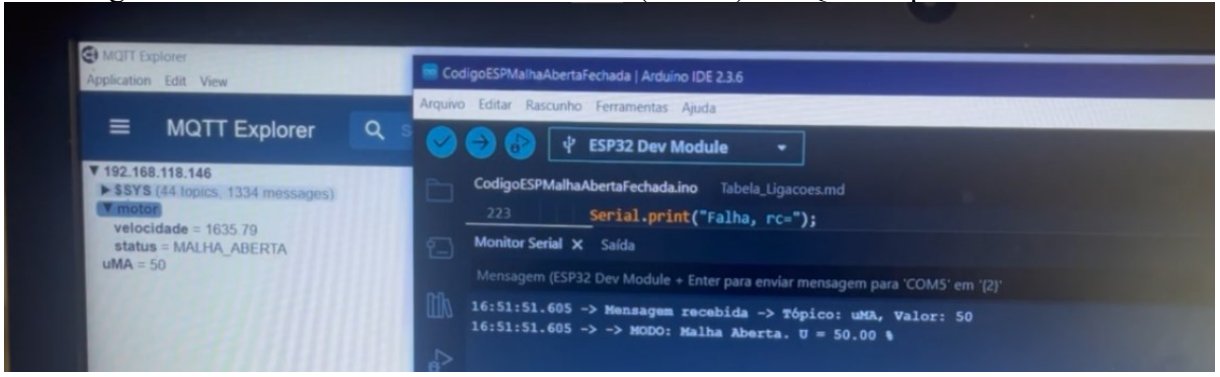


Figura 34 – Malha Aberta - Monitor Serial (ESP32) e MQTT Explorer



O mesmo é feito para o módulo de Malha Fechada onde incluímos a referência de referência de 1000 RPM e ganho proporcional de 0,3, conforme evidenciado na Figura 35 e a validação desses dados no Monitor Serial e no MQTT Explorer, mostrado na Figura 36.

Figura 35 - Input de dados Malha Fechada

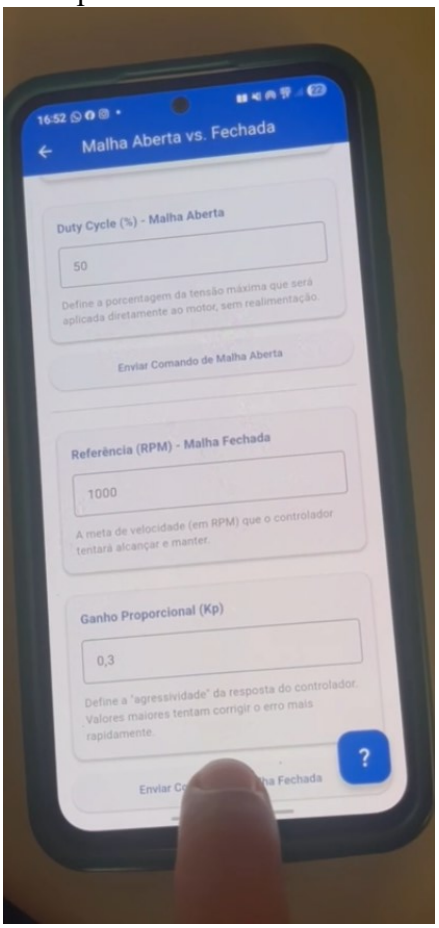
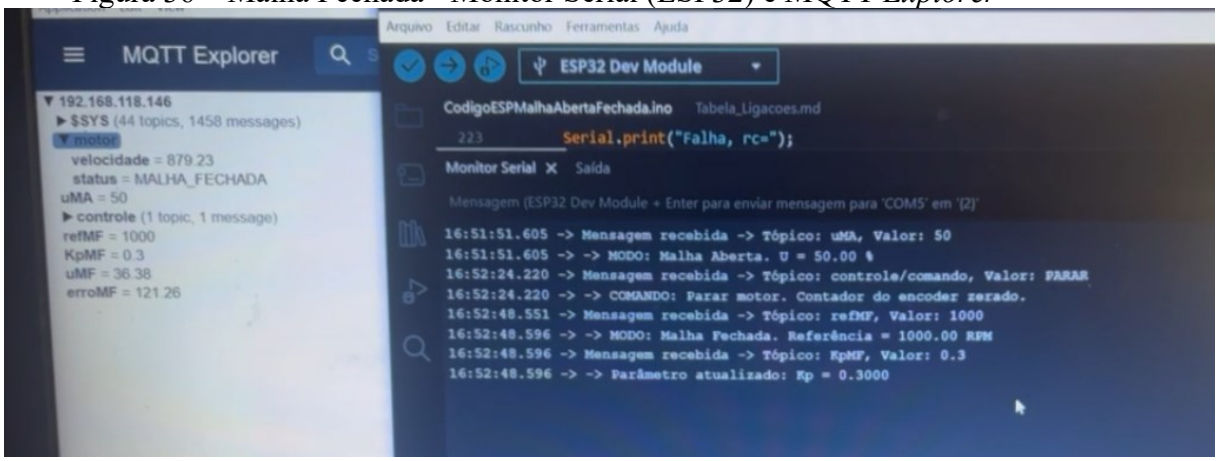


Figura 36 – Malha Fechada - Monitor Serial (ESP32) e MQTT *Explorer*



Alterando os dados em malha fechada, observamos nas Figuras 37 e 38 um comportamento bem condizente com a teoria, o que demonstra a confiabilidade na solução idealizada.

Figura 37 - Alteração dos dados em Malha Fechada

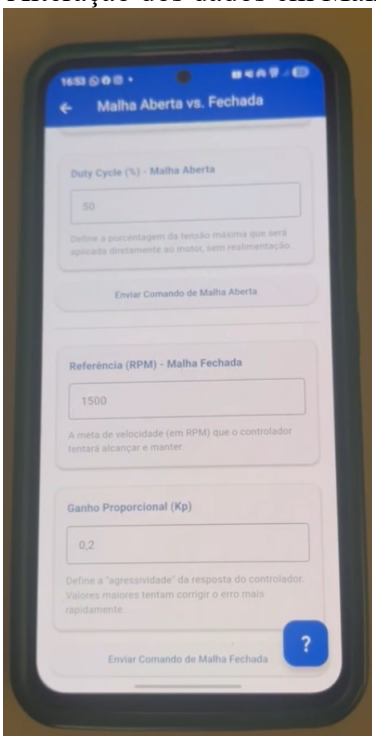
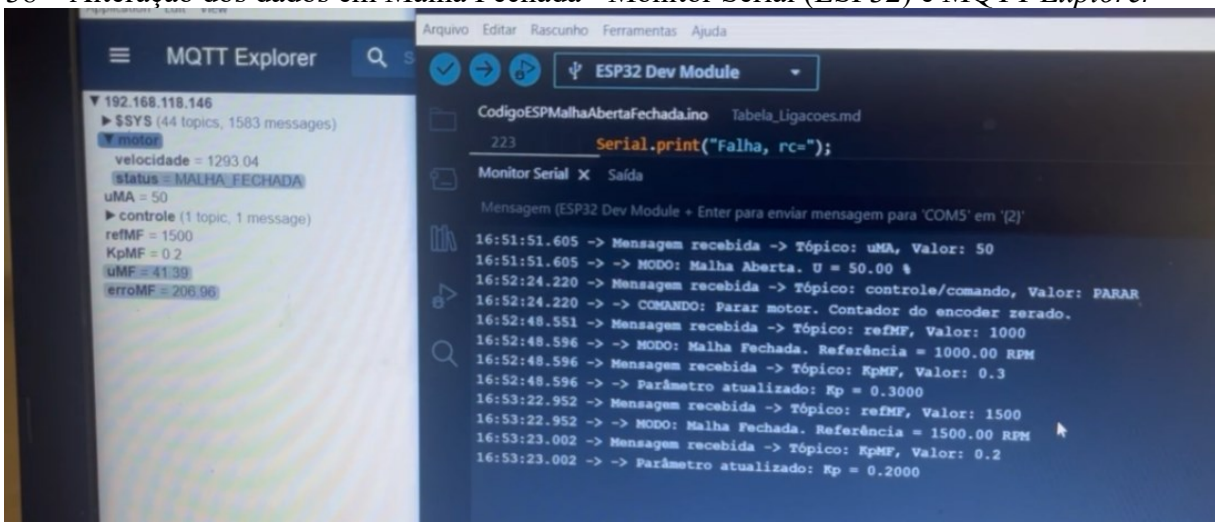


Figura 38 – Alteração dos dados em Malha Fechada - Monitor Serial (ESP32) e MQTT *Explorer*



Para uma melhor visualização do comportamento do protótipo, o vídeo com a implementação do projeto foi gravado e disponibilizado no Git Hub[1] do MAPL, bem como todos os códigos utilizados na implementação.

5 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo principal o desenvolvimento e a validação de uma solução completa, baseada na arquitetura da Internet das Coisas Industrial (IIoT), para viabilizar a execução remota das aulas práticas da disciplina de Controle de Sistemas Lineares, com foco no laboratório de "Malha Aberta vs Malha Fechada". A motivação central foi superar as barreiras do ensino a distância para atividades que demandam interação com equipamentos físicos, além de modernizar a abordagem pedagógica com tecnologias alinhadas à Indústria 4.0.

Pode-se afirmar que os objetivos propostos foram plenamente alcançados. Foi desenvolvido um "ecossistema" funcional composto por um *firmware* para o microcontrolador ESP32, programado em C/C++, e um aplicativo móvel para dispositivos *Android*, desenvolvido com o *framework Flutter* e a linguagem *Dart*. A comunicação entre as partes, intermediada por um *broker* MQTT, provou-se robusta e com tempo de resposta adequado para as atividades de controle propostas. O aplicativo móvel demonstrou ser uma Interface Homem-Máquina (IHM) eficaz, permitindo ao aluno não apenas conectar-se ao sistema e selecionar o experimento, mas também enviar parâmetros de controle para os modos de malha aberta e malha fechada, além de monitorar em tempo real variáveis essenciais como velocidade, sinal de controle e erro.

O sucesso na implementação e nos testes práticos valida a arquitetura proposta como uma solução viável e de baixo custo para laboratórios didáticos remotos. A utilização de ferramentas modernas e de código aberto, como *Flutter*, a biblioteca *PubSubClient* e o protocolo MQTT, demonstra a acessibilidade de se criar sistemas de controle distribuídos. A principal contribuição deste trabalho é, portanto, a criação de uma ferramenta pedagógica que não apenas soluciona o problema do acesso remoto, mas também enriquece a formação do estudante de engenharia, colocando-o em contato direto com os pilares tecnológicos da automação e da conectividade industrial contemporânea. A integração de um visualizador de PDF com o material teórico diretamente no aplicativo reforça ainda mais seu valor como uma plataforma de aprendizado completa.

Apesar do êxito, reconhecem-se algumas limitações e pontos para desenvolvimento futuro. O sistema atual depende da estabilidade da rede *Wi-Fi* local, e variações na latência poderiam impactar experimentos que exijam um controle em tempo real mais rigoroso. Além disso, o controlador de malha fechada implementado foi o Proporcional (P), que, embora suficiente para os objetivos didáticos desta prática, serve como ponto de partida para lógicas mais complexas.

6 TRABALHOS FUTUROS

Mesmo atingindo os objetivos propostos durante a idealização e implementação do projeto fomos apresentados a melhorias que podem ser exploradas futuramente, como:

- Implementação de alteração dos códigos do ESP32 remotamente, pelo professor a cada mudança de módulo do laboratório, que hoje acontece a cada 15 dias.
- A criação de uma bancada única, com outros protótipos a fim de verificar outras implementações possíveis da teoria estudada.
- A inclusão de gráficos em tempo real no aplicativo para a visualização da resposta do sistema (como a resposta ao degrau) enriqueceria imensamente a análise do aluno.
- A geração do relatório pré estruturado pelo professor, em arquivo .pdf para que receba as informações analisadas pelo aluno de forma ágil e prática.
- A integração com um *broker* em nuvem poderia ser explorada para permitir o acesso aos experimentos de fora da rede da universidade, ampliando ainda mais o alcance e a flexibilidade da solução.
- Implementação dos outros exercícios práticos da aula de Controle Linear.

7 REFERÊNCIAS

Assunção, E. Teixeira, M. **Controle Linear I. Parte A – Sistemas Contínuos no Tempo**. 2013. Disponível em: https://www.feis.unesp.br/Home/departamentos/engenhariaeletrica/lpc1672/apostila-decontrole-linear-i.pdf. Acesso em 25 ago. 2025.

AUGUSTO, P. **Laboratório 2 – Malha Aberta vs Malha Fechada**. Uberlândia, 2022.

Flutter um framework para desenvolvimento mobile. **RECIMA21**, v. 3, n. 11, 2022. Disponível em: https://recima21.com.br/index.php/recima21/article/view/2230 . Acesso em 27 ago. 2025.

MASCHIETTO, Luís Gustavo et al. **Arquitetura e infraestrutura de IoT**. Porto Alegre: SAGAH, 2021. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9786556901947. Acesso em 7 ago. 2025.

OGATA, Katsuhiko. **Engenharia de Controle Moderno**. 5. ed. São Paulo: Pearson Prentice Hall, 2010.

TAVARES, J. J.-P. Z. D. S. **Arquitetura, Comunicação e Protocolos IoT e IIoT**. Uberlândia, 2025.

YADAV, Rajesh Kumar. The untold story of Visual Studio Code: **A revolution in software development**. Dev.to, 2023. Disponível em: https://dev.to/rajeshkumaryadavdotcom/the-untold-story-of-visual-studio-code-a-revolution-in-software-development-44pp. Acesso em 27 ago. 2025.

ANEXO A – ROTEIRO LABORATÓRIO CONTROLE LINEAR

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

FEMEC 42060

CONTROLE DE SISTEMAS LINEARES

Relatório de Laboratório 2 - Malha Aberta *vs* Malha Fechada

Prof. Pedro Augusto

19 de maio de 2022

Relatório de Laboratório 2 - Malha Aberta *vs* Malha Fechada Prof.

1 OBJETIVOS

Neste laboratório comparar-se-ão as estruturas de controle em malha aberta e em malha fechada.

2 INTRODUÇÃO

Controlar um sistema pode ser definindo como impor um comportamento desejado a um processo. Tipicamente isso envolve guiar a saída para o entorno de uma referência com determinados requisitos de velocidade e precisão. Com esse propósito, é possível adotar estruturas de controle em malha aberta (MA) ou em malha fechada (MF).

Em MA, manipula-se diretamente a entrada na planta para resultar na saída desejada. Para que isso funcione adequadamente, a relação entrada-saída deve ser conhecida com precisão, não podendo haver descasamentos de modelo ou perturbações externas. Um diagrama de blocos do controle em MA é mostrado na Figura 1.

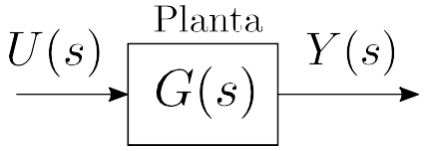


Figura 1: Estrutura de controle em malha aberta.

Algumas vantagens dessa estrutura são simplicidade de construção e manutenção, implementação barata e conveniência em situações nas quais não se é possível medir a saída do processo. Em contrapartida, os efeitos de perturbações ou descasamentos de modelo afetam diretamente a saída da planta. Mais ainda, para manter um desempenho adequado, é possível que uma recalibração frequente seja necessária.

Já em MF (Figura 2) as informações da saída são levadas em consideração no cálculo da entrada (realimentação). Em malha fechada, a variável manipulada pelo operador é o valor desejado para a saída. O controlador é responsável por calcular a entrada de acordo com o erro de rastreamento. Com efeito, tem-se uma maior robustez a perturbações externas e descasamentos de modelo. Por outro lado, o custo de implementação desse controlador é maior, bem como a complexidade de manutenção.

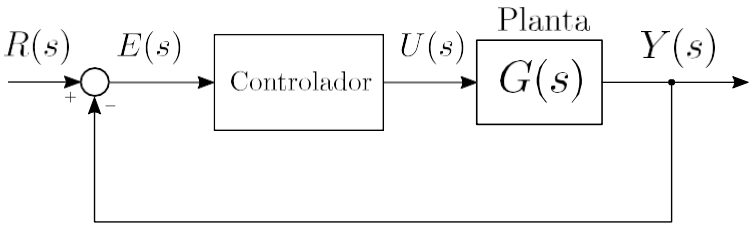


Figura 2: Estrutura de controle em malha fechada.

No que se segue, as estruturas de controle em MA e MF serão comparadas experimentalmente.

3 LISTA DE MATERIAIS

Os materiais para realização do presente laboratório são listados abaixo

- Arduino UNO
- Fios de conexão
- Fonte DC
- Motor DC
- Ponte H - L298N
- *Protoboard*

4 PROCEDIMENTO EXPERIMENTAL

Na sequência circuitos eletrônicos e códigos para implementação de controladores em MA e MF serão mostrados.

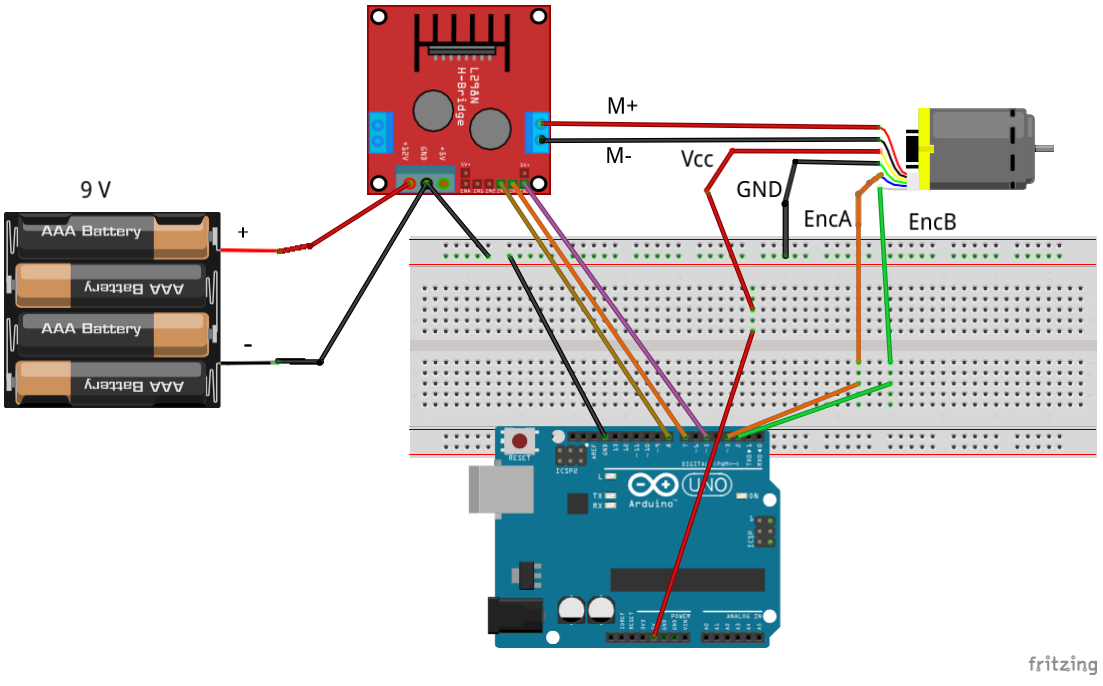
4.1 Controle de sistemas em malha aberta

Neste item, variar-se-á a entrada da planta (ciclo ativo de onda PWM) e monitorar-se-á a velocidade do motor. A partir dessas informações, um sistema de controlador em MA será implementado.

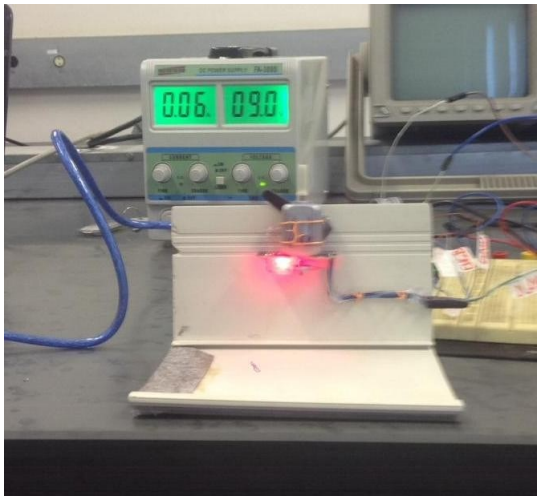
- Faça download da biblioteca Encoder.h em https://github.com/PaulStoffregen/Encoder e inclua a biblioteca em “Sketch”-> “Incluir Biblioteca”-> “Adicionar Biblioteca ZIP”

- Monte seguinte o circuito:

7.1.1 MUITA ATENÇÃO NA CONEXÃO DOS CABOS VCC E GND DO ENCODER!!!



- Posicione o motor na posição indicada na Figura 4.1



- Utilize o código abaixo para variar o *duty cycle* de entrada e medir a velocidade do motor

```
//Incluindo biblioteca para leitura do encoder
#include <Encoder.h>


// Definindo objeto meu Encoder
Encoder meuEncoder(2, 3);

//Definindo outras variaveis uteis
double velAng, theta = 0.0, thetaAnt = 0.0, tempo1 = 0.0,
   tempo2 = 0.0, dt, u;
long contEnc = 0.0;

//variaveis da media movel
double vel1 = 0.0, vel2 = 0.0, vel3 = 0.0, mediavel;
void setup() {

  // Inicializando comunicacao serial
  Serial.begin(115200);

  //Definindo Entradas da ponte H pin
  Mode(5,OUTPUT); //velocidade de giro

  pinMode(7,OUTPUT); //sentido de giro pin
  Mode(8,OUTPUT); //sentido de giro
```

```
  //sentido  horario
  digitalWrite    (7    ,    HIGH    );
  digitalWrite (8, LOW );
}


void loop() {
  //Salvando valores anteriores de tempo e posicao tempo1
  = tempo2 ;
  thetaAnt  =  theta;


  //Determinando leitura autal do encoder contEnc
  =  meuEncoder. read ();


  //Calculando theta a partir da leitura do encoder theta
  = contEnc *360/(334*4); //resolucao do encoder e
      334


  //Calculando  diferenca  de  tempo  ente  instante  atual   e
     instante  da  ultima  leitura
  dt = tempo2 - tempo1; //em micro s
  //Calculando  velocidade  angular                          rpm

  //calculando  media  movel  dos  tres  ultimos  valores
  vel1   =   vel2   ;
  vel2   =   vel3   ;
  vel3 = velAng;
  mediavel = (vel1 + vel2 + vel3)/3.0;


  //Variando  duty  cycle
  u = 0.0;
  analog Write (5 ,255.0* u /100.0);


  //Imprimindo na porta serial
  Serial.   print(   mediavel);
  Serial. print("     "); Serial.
  print(u); Serial. print(" ");
  Serial. println (tempo2 /1000000);// em  s
}
```

- Modifique o código acima para preencher a tabela abaixo.

  Nota 1: os valores de velocidade podem ser obtidos monitorando a porta serial: "Ferramentas"->"Monitor serial" ou Ctrl+Shift+M, ou "Ferramentas"->"*Plotter* serial" ou Ctrl+Shift+L

Tabela 1: Relação entre *duty cycle* e temperatura

duty cycle %	*y* (*rpm*)
0	0
25	0
35	825
50	2345
75	3790
85	4180

Nota 2: Em valores de *duty cycle* muito baixos, pode ser necessário partir manualmente o motor devido à presença de atrito seco (dinâmica não linear que negligenciaremos)

- Construa um gráfico entre velocidade (rpm) e *duty cycle* (%) e determine a equação que relaciona essas grandezas (**regressão linear**)

- Utilizando a relação velocidade-*duty cycle* obtida, complete a tabela abaixo

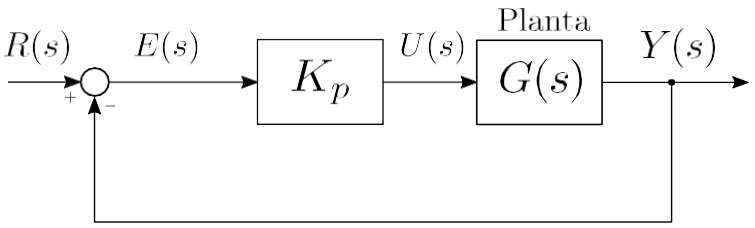
Tabela 2: Desempenho de sistema de controle em MA

Referência (rpm)	*duty cycle* (%)	*y* (rpm) - valor medido
1000	30	0
1500	38,93	1400
4000	83,57	4130

- Salve os dados para responder à Seção 1 do relatório

4.2 Controlador proporcional

- Altere o código apresentado nessa seção para implementar controlador propor- cional em MF conforme ilustrado na figura a seguir.




```
  //Incluindo biblioteca para leitura do encoder
#include <Encoder.h>

// Definindo objeto meu Encoder
Encoder meuEncoder(2, 3);

//Definindo outras variaveis uteis
double velAng, theta = 0.0, thetaAnt = 0.0, tempo1 = 0.0,
   tempo2 = 0.0, dt, u;
long contEnc = 0.0;

// Variaveis da media movel
double vel1 = 0.0, vel2 = 0.0, vel3 = 0.0, mediavel;

// Referencia
double Kp, erro = 0.0, ref = 0.0;

// Integral do erro double
erro_int;

void setup() {
  // Inicializando comunicacao serial
  Serial.begin(115200);

  //Definindo Entradas da ponte H pin
  Mode(5,OUTPUT); //velocidade de giro pin
  Mode(7, OUTPUT); //sentido de giro pin
```

```
  digitalWrite(7,HIGH);  digitalWrite(8,LOW);
}


void loop() {
  //Salvando valores anteriores de tempo e posicao
  tempo1 = tempo2;
  thetaAnt = theta;

  //Determinando leitura autal do encoder contEnc
  = meuEncoder.read();

  //Calculando theta a partir da leitura do encoder
  theta = contEnc*360/(334*4);

  //Calculando diferenca de tempo ente instante atual  e
     instante da ultima leitura
  dt = tempo2 - tempo1;//em micro s
  //Calculando velocidade angular                    rpm

  //calculando media movel dos tres ultimos valores
  vel1 = vel2; vel2 = vel3; vel3 = velAng; mediavel
  = (vel1 + vel2 + vel3)/3.0;


  //tempo do degrau
  if(tempo2/1000000 >= XXXX){ ref
    = XXXX; //rpm
  }


  // Calculando erro de rastreamento erro
  = xxxxxx;


  // Implementando lei de controle proporcional
  u = xxxxxx;                    entre 0 e 90
  u = min(u, 90.0);     u = max(u, 0.0);

  // Enviando controle saturado para porta analogica
  analogWrite(5, 255.0*u/100.0);
```

```
  //Imprimindo na porta serial
  Serial.  print(  mediavel);
  Serial. print("    "); Serial.
  print(u); Serial. print(" ");
  Serial. println(tempo2 /1000000);// em  s
}
```

- Verifique o comportamento da planta para um degrau na referência de 1000 **rpm** após 2 s com K_p = 0,1, K_p = 0,25, K_p = 0,5 **e** K_p = 1

- Para cada ensaio, salve os dados na porta serial ("Ferramentas"->"Monitor serial") em um arquivo .txt

  Nota: deve-se copiar todos os dados lidos na porta serial (Ctrl+A e Ctrl+c) e colá-los em um arquivo de .txt (Ctrl+v). **Desconectar o cabo USB de alimentação do Arduino pode ser útil nesse processo**

- Observe os gráficos de velocidade e controle. Com esse propósito, sugere-se utilizar o código abaixo para gerar as imagens

```
clear, close all, clc
%Carregando  dados load
(' Nome Do Arquivo . txt')
Data = NomeDoArquivo;

%Criando figura
figure
subplot(1,2,1)
%  Plotando  dados  de  velocidade plot(
Data(:,3), Data(:,1),'b-','LineWidth',2) grid
on, hold on

%Ajustando nome dos eixos e tamanho de letra xlabel('t
(s)'); ylabel('y (rpm)');
xlim([0  max(Data(:,3))])

subplot(1,2,2)
% Plotando dados de entrada
```

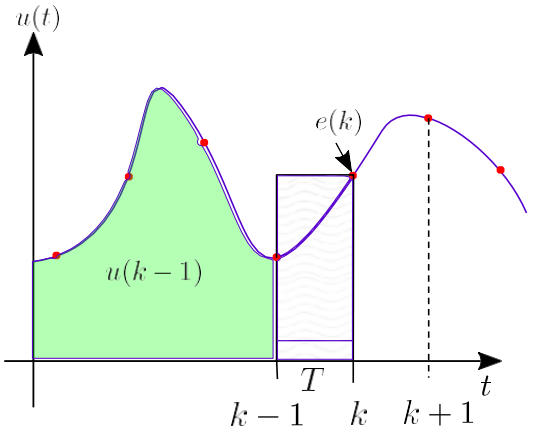
```
grid on, hold on

%Ajustando nome dos eixos e tamanho de letra xlabel('t
(s)');  ylabel('u  (\%)');
xlim([0  max(Data(:,3))])
```

- Salve os dados para responder à Seção 2 do relatório

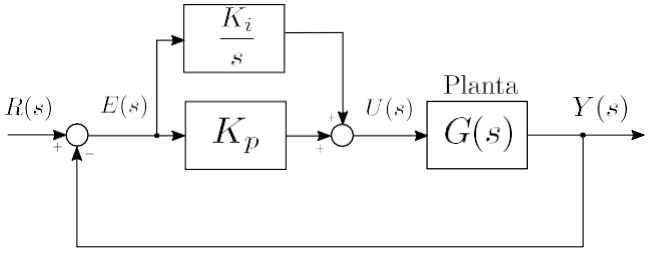
4.3 Exercício

- A integral de um sinal $u(t) = \int_0^{\infty} e(t)dt$ pode ser realizada a tempo discreto da seguinte forma

$$u(k) = u(k-1) + e(k)\Delta t \qquad (1)$$



- Utilizando essa aproximação numérica, implemente a lei de controle ilustrada a seguir



7.1.2 Lembre-se: a variável dt está em microssegundos!!

- Ajuste K_p = 0,15 e K_i = 0,08
- Altere a referência para **3000 rpm** e verifique a resposta da planta
- Salve os dados para responder à Seção 3 do relatório.