UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

STÉFANO CAMPOS DE OLIVEIRA

DESENVOLVIMENTO DE UMA INTERFACE DE COMANDO PARA MÁQUINA DE RAIO X

Joinville
2022

STÉFANO CAMPOS DE OLIVEIRA

DESENVOLVIMENTO DE UMA INTERFACE DE COMANDO PARA MÁQUINA DE RAIO X

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica no curso de Engenharia Mecatrônica, da Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville.

Orientador: Prof. Dr. Anderson Wedderhoff Spengler

Coorientador: Alexandre Ferrardo
Coorientador: Erick Araujo Nunes

Joinville
2022

Dedico este trabalho a minha família, ao meu vô Nelcy e ao Vicente.

AGRADECIMENTOS

Agradeço imensamente minha família por fazer parte da minha vida, me incentivar e apoiar durante todos meus percursos pois sem vocês, não conseguiria. Ao meus pais, Carlos e Eliane agradeço por tudo que sou, onde estou e estarei por toda minha vida. Agradeço também ao meu maninho Victor por ser minha inspiração e parte de quem eu sou. À minha cunhada Maria Júlia, agradeço por ser uma irmã para mim. Amo vocês.

Mariana, te agradeço por estares sempre comigo em todos os momentos bons e difíceis desse meu caminho. Agradeço pelo amor, carinho e paciência que tu tens e teve comigo em todo esse tempo. Minha parceira, sem você não conseguiria. Te amo.

Ao meu vô Nelcy e ao meu tio Vicente, aonde quer que estejam, agradeço por sempre terem acreditado em mim. Podem não estar mais aqui, mas sempre estarão comigo, seu engenheiro.

Gostaria de agradecer a um grupo de amigos espetacular que tenho, os Bojackos. Obrigado por todos os momentos que passamos na graduação, vocês tornaram essa experiência única. Agradeço também à Caverna (Anti)Social por terem me acolhido como membro. Parafraseando ZILLI, Vinicius: “Amigos, vocês tornaram os anos em Joinville tão agradáveis e com memórias que perdurarão a existência.”.

Ao Lucas de Camargo, agradeço muito por ter sido meu camarada, minha dupla e guru dos códigos em vários momentos da nossa graduação. Obrigado também por toda paciência.

À Siemens Healthineers agradeço imensamente por tudo ensinado e por tudo que cresci como profissional desde que ingressei no estágio. Agradeço ao Alexandre por ser um gestor e líder excepcional que sempre me incentivou à excelência e a ser um profissional cada vez melhor. Agradeço também ao Engenheiro Erick por ser meu tutor e colega, que me ensinou tudo, levou ao caminho do conhecimento e sempre esteve ao meu lado, você é *big*.

“If the hand be held between the discharge-tube and the screen, the darker shadow of the bones is seen within the slightly dark shadow-image of the hand itself...For brevity’s sake I shall use the expression “rays”; and to distinguish them from others of this name I shall call them “X-rays”.(RÖNTGEN, 1895).

RESUMO

A geração de raios X em máquinas radiológicas médicas precisa ser acionada e controlada de maneira simples, segura e robusta para que se produza um resultado confiável em diagnósticos e que seja saudável para o indivíduo exposto. Hoje a parametrização e disparo de raio X são feitos por meio de uma placa dedicada externa ao computador do operador. Este trabalho propõe centralizar em um computador o acionamento e controle do gerador de raios X para aumentar a flexibilidade da aplicação e torná-lo intuitivo ao operador. Para tal foi desenvolvido um programa de computador e interface de usuário para parametrizar e acionar o gerador de raios X. O *hardware* base utilizado foi um microcontrolador ESP32 e a metodologia seguida foi o desenvolvimento baseados em diagramas UML (do inglês Unified Modeling Language). Foram atingidos os requisitos propostos nesse projeto, tanto de *software* quanto de *hardware*.

Palavras-chave: Raio X. Interface de acionamento. Parametrização. Gerador de raios X.

ABSTRACT

The generation of X-rays in medical radiological machines needs to be activated and controlled in a simple, safe and robust way so that a reliable diagnostic result is produced and that it is healthy for the exposed individual. Today, parameterization and X-ray shooting are done through a dedicated board external to the operator's computer. This work proposes to centralize the activation and control of the X-ray generator on a computer to increase application flexibility and make it intuitive for the operator. For this, a computer program and user interface was developed to parameterize and activate the X-ray generator. The base hardware used was an ESP32 microcontroller and the methodology followed was the development based on UML (Unified Modeling Language) diagrams. The requirements proposed in this project, both software and hardware, were fulfilled.

Keywords: X-ray. Drive interface. Parameterization. X-ray generator.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

f	Frequência
λ	Comprimento de onda
h	Constante de Plank
KE	Energia Cinética
ϕ	Função trabalho
v_0	Frequência crítica
V_0	Tensão crítica
Fm	Frequência máxima PWM
V_{out}	Tensão de saída
Vcc	Tensão corrente contínua

SUMÁRIO

1 INTRODUÇÃO

O raio X foi descoberto em 1895 pelo físico e engenheiro mecânico Wilhelm Conrad Röntgen, quando estava realizando experimentos com tubos catódicos e percebeu, por acaso, uma luminescência em uma das telas fluorescentes ao serem atingidas pela luz emitida do tubo. O fenômeno persistiu mesmo ao colocar a tela dentro de uma caixa de papelão, o que o fez perceber que o tubo não estava emitindo apenas luz visível, mas um outro tipo de radiação que chamou de raio X pela sua natureza, então, desconhecida (SUETENS, 2009). No caso do experimento de Röntgen, o material que começou a brilhar foi o platinocianeto de bário (NAJARIAN; SPLINTER, 2012).

Röntgen percebeu que o raio X era atenuado em diversas maneiras por diversos materiais e que, assim como a luz, poderia ser capturado em uma placa fotográfica. Assim que descobriu o raio X, Röntgen capturou a primeira imagem de uma mão humana, a de sua esposa. A partir desse momento abriram-se portas para a exploração desse recém descoberto raio X para aplicações em medicina, dentre outras áreas. Em poucos meses as radiografias já estavam sendo usadas em práticas clínicas, o potencial para diagnóstico de imagens, dentro do corpo humano, foi quase que imediatamente reconhecido como o principal uso da nova forma de radiação, e ainda hoje é um dos meios mais utilizados de diagnóstico por imagem (NAJARIAN; SPLINTER, 2012).

O princípio da radiação do raio X se baseia na desaceleração de elétrons durante a interação com os núcleos atômicos, principalmente de metais pesados, desaceleração essa que transforma a energia cinética dos elétrons em outras formas de energia, como energia térmica e energia eletromagnética (NAJARIAN; SPLINTER, 2012). Essa última corresponde ao fenômeno do raio X, o qual possui uma característica de comprimento de onda curto, na ordem de Angstrøms ($10^{-10}m$) (SUETENS, 2009).

Embora as máquina de raio X tenham evoluído muito desde sua invenção, o princípio de funcionamento continua o mesmo: o raio X é gerado dentro de um tubo de raio X, que é mantido à vácuo com um cátodo e ânodo. O cátodo libera uma nuvem de elétrons por excitação térmica, onde são acelerados em direção ao ânodo por diferença de potencial e ao colidirem com os átomos do ânodo, desaceleram e geram as ondas eletromagnéticas do raio X (NAJARIAN; SPLINTER, 2012).

As primeiras máquinas de raio X eram acionadas apenas com energia elétrica e, com o tempo, foram aperfeiçoadas para um controle mais refinado com a eletrônica. As máquinas acompanharam a necessidade da digitalização tanto no acionamento do gerador, com o advento de interfaces de comandos dedicadas, quanto na detecção do raio X para gerar imagens, que antes era feita através de filmes sensíveis ao espectro

de raio X e hoje são mais comumente feitas através de detectores digitais (SHUNG; SMITH; TSUI, 2012).

A digitalização das máquinas de raio X traz mais possibilidades de ações para a operação, manutenção, usabilidade e conectividade entre máquinas, supervisórios e usuários. No âmbito da operação, a digitalização proporciona mais facilidade no uso e possibilidades de rotinas cada vez mais automatizadas, como rotinas de exames específicos com parâmetros mais adequados, tornando-os assim menos dependente da escolha do técnico operador para se obter a melhor qualidade de imagem, com o menor tempo de exposição possível para o paciente (NAJARIAN; SPLINTER, 2012).

No equipamento radiológico que está sendo estudado, não há ainda uma interface centralizada em um computador para controlar o acionamento do gerador de raio X. Tem-se hoje um painel de comando desanexado do computador que processa as imagens (em um conjunto digital).

Propõe-se, portanto, uma pesquisa para centralizar os comandos de parametrização e ativação do gerador de raio X através de uma interface gráfica de usuário, em um único programa de computador, conectado a um *hardware* intermediário que adequa as informações para o envio ao gerador em uma sequência correta. Com um sistema em pleno funcionamento será possível acionar de forma segura e fácil o gerador, além de enviar a melhor dose de radiação possível ao paciente para determinado exame. A centralização dos comandos pode, ainda, facilitar o diagnóstico de manutenções e abrir oportunidades de conectividade entre equipamentos, aumentando a possibilidade de outras funcionalidades do sistema.

1.1 OBJETIVO

A fim de resolver a descentralização dos comandos de conjuntos radiológicos, apresentam-se os objetivos gerais e específicos deste desenvolvimento nessa seção.

1.1.1 Objetivo Geral

Desenvolver uma interface de comando centralizada para um conjunto radiológico a fim de enviar informações de parametrização e ativação do gerador de raio X.

1.1.2 Objetivos Específicos

- Realizar um levantamento de entradas e saídas do gerador de raio X;
- Definir o *hardware* a ser utilizado para o desenvolvimento;
- Implementar algoritmos para o envio de sinais ao gerador;
- Criar interface de usuário;
- Validar os resultados de forma experimental em um osciloscópio.

2 FUNDAMENTAÇÃO TEÓRICA

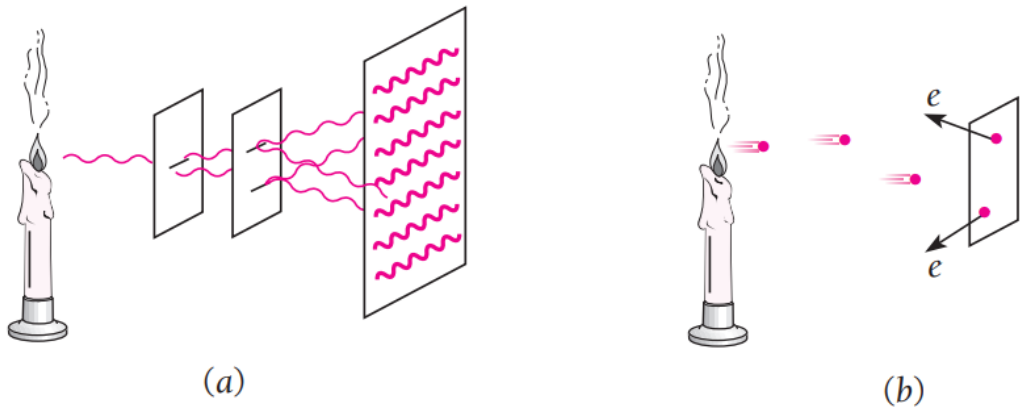
Neste capítulo serão apresentados alguns conceitos fundamentais para o entendimento de fenômenos físicos que contribuem para a geração de raio X. Na Seção 2.1 serão abordados os princípios físicos da luz como onda e partícula e do fóton. Também serão apresentados conceitos como o efeito fotoelétrico e emissão termoiônica nas seções 2.1.1 e 2.1.2 respectivamente.

Com os conceitos básicos já entendidos, será apresentada a física do fenômeno do raio X na Seção 2.2 e na Seção 2.3 o funcionamento da geração de raio X, contemplando o gerador e tubo de raio X. Por fim serão exibidos alguns equipamentos similares que já possuem uma interface de comando para que seja possível ter uma referência de projeto.

2.1 FÓTON E LUZ

De acordo com a teoria ondulatória, a luz deixa a fonte com sua energia continuamente espalhada através do padrão de onda, entretanto, de acordo com a teoria quântica, a luz consiste em pequenos pacotes de energia chamados de fótons, cada qual pequeno o suficiente para ser absorvido por um único elétron (BEISER, 2003). A Figura 1 ilustra o comportamento da luz sob duas visões diferentes de teorias: a teoria ondulatória da luz explica a difração e a interferência que a teoria quântica não pode explicar enquanto a teoria quântica explica o efeito fotoelétrico, que a teoria ondulatória não pode explicar. Com a descoberta do efeito fotoelétrico, que será explicado nessa seção, foi mostrado que a teoria da luz como onda e a teoria quântica da luz se complementam, portando, a luz é tanto onda quanto partícula (BEISER, 2003).

Figura 1 – (a) Teoria ondulatória da luz. (b) Teoria quântica da luz.



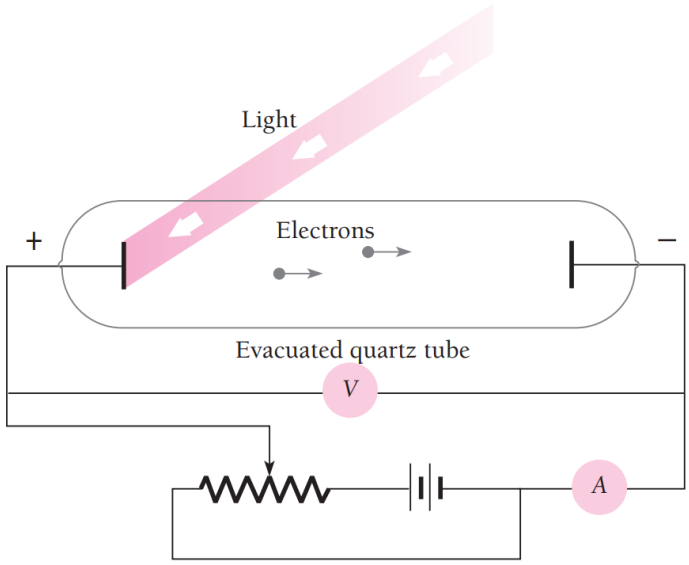
Fonte: Beiser (2003, p.67)

2.1.1 Efeito fotoelétrico

Durante seus experimentos com geração e detecção de radiação eletromagnética, Heinrich Hertz percebeu que ao retirar a luz do ambiente de experimento para que fosse possível observar as faíscas entre as placas, a intensidade das faíscas diminuía, e ao adicionar mais luz ao ambiente a frequência de faíscas aumentava. Então ele decidiu incidir luz ultra violeta que possui alta frequência e o resultado da emissão de faíscas foi ainda mais frequente, logo descobriram que a causa desse efeito era a ocorrência da emissão de elétrons quando a frequência da luz era suficientemente alta (BEISER, 2003). Esse efeito foi chamado de efeito fotoelétrico e os elétrons emitidos são chamados de fotoelétrons.

A Figura 2 mostra como o efeito fotoelétrico foi estudado: um tubo selado contém dois eletrodos conectados a uma fonte de tensão variável, uma luz ultravioleta é emitida no lado do ânodo então os elétrons do átomo da placa de ânodo são agitados e vibram pelo campo elétrico oscilante da radiação incidente. Alguns desses fotoelétrons têm energia suficiente para se desprender do elétron e alcançar o cátodo, apesar de sua polaridade negativa (BEISER, 2003), e assim consegue-se obter uma leitura de corrente elétrica decorrente do movimento desses fotoelétrons.

Figura 2 – Observação do efeito fotoelétrico



Fonte: Beiser (2003, p.63)

Os elétrons mais próximos ao núcleo requerem mais energia para se desprenderem ou se tornarem livres, enquanto os elétrons nas camadas elétricas mais distantes necessitam de menos energia. A quantidade de energia necessária é determinada pelo número atômico do elemento (SHUNG; SMITH; TSUI, 2012) e quando a tensão nos terminais é aumentada a um certo nível V_0, não há mais chegadas de fotoelétrons no terminal do cátodo, isso é indicado pela queda de corrente

elétrica medida e essa tensão V_0 corresponde à máxima energia cinética do fotoelétron (BEISER, 2003).

A luz incidente como onda carrega uma energia potencial que é parte absorvida pelo metal e parte transformada em energia cinética, o que explica a existência do efeito fotoelétrico. Porém de acordo com (BEISER, 2003), três descobertas experimentais mostram que o efeito elétrico não pode ser totalmente explicado com a teoria da luz como uma onda eletromagnética.

Em 1905 Einstein percebe que o efeito fotoelétrico poderia ser explicado caso a energia da luz não se espalhasse apenas como uma onda eletromagnética, mas também como pequenos pacotes de energia chamados de fótons em que cada fóton de luz com frequência v teria a energia hv, onde h é a constante de Plank indo ao encontro da energia quântica de Plank da teoria quântica da luz como onda eletromagnética (BEISER, 2003). Essa quebra radical na explicação da luz foi proposta por Einstein em *Proposal of the Photon Concept* como segue:

> Parece-me que as observações associadas à radiação de corpo negro, fluorescência, produção de raios catódicos por luz ultravioleta e outros fenômenos relacionados com a emissão ou transformação da luz são mais facilmente compreendidos se assumirmos que a energia da luz é distribuída de forma descontínua no espaço. De acordo com a suposição a ser considerada aqui, a energia de um raio de luz que se espalha a partir de uma fonte pontual não é continuamente distribuída em um espaço crescente, mas consiste em um número finito de quanta de energia que estão localizados em pontos no espaço, que se movem sem divisão, e que só pode ser produzido e absorvido como unidades completas (EINSTEIN, 1965, p. 2, tradução nossa).

De acordo com (BEISER, 2003) as três observações a seguir corroboram a hipótese de Einstein:

a) Pelo motivo da energia da onda eletromagnética ser concentrada nos fótons e não espalhada, não deve ter atrasos na emissão de fotoelétrons.
b) Todos os fótons de frequência v tem a mesma energia, então mudando a intensidade da luz monocromática mudará a quantidade de fotoelétrons, mas não a energia dos mesmos.
c) Quanto maior a frequência v, maior a energia hv do fóton.

A energia mínima ϕ para que um elétron consiga escapar de uma partícula de metal é dada pela função trabalho do metal e é relacionada com a frequência crítica v_0 pela Equação 1 (BEISER, 2003).

$$\boldsymbol{\phi} = h\boldsymbol{v}_0 \tag{1}$$

Portanto, quanto maior a função trabalho, maior a energia necessária para que um elétron se desprenda de uma superfície metálica e maior é a frequência crítica para que o efeito fotoelétrico aconteça. De acordo com Einstein o efeito fotoelétrico é dado

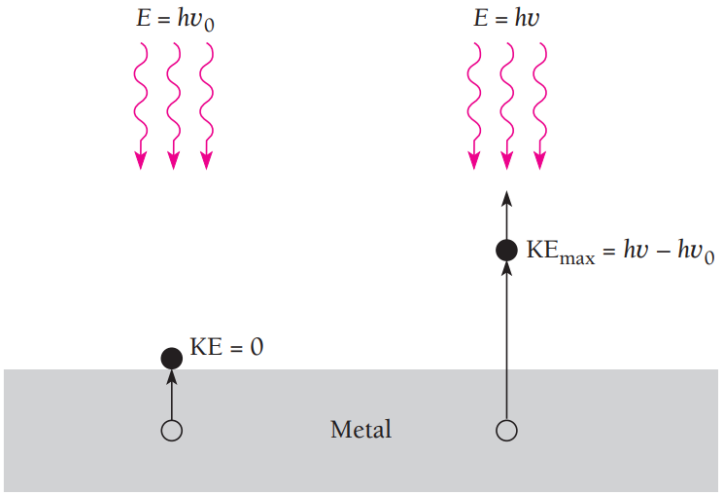
pela Equação 2 a seguir (BEISER, 2003):

$$h\boldsymbol{v} = KE_{max} + \boldsymbol{\phi} \tag{2}$$

onde hv é a energia do fóton, KE_{max} é a energia fotoelétrica máxima, a qual é proporcional ao potencial de parada (BEISER, 2003). A Equação 2 pode ser reescrita de acordo com a Equação 3 e pode ser ilustrada com a Figura 3.

$$KE_{max} = h(\boldsymbol{v} - \boldsymbol{v}_0) \tag{3}$$

Figura 3 – Energia cinética do efeito fotoelétrico.



Fonte: Beiser (2003, p.66)

A explicação desse efeito é essencial para o entendimento da física por trás do raio X. Muitos nomes como Maxwell, Plank, Hertz e Einstein contribuíram para tornar possíveis tantas tecnologias presentes hoje, dentre as quais, a máquina de raio X usada largamente em diversas áreas. A explicação desse efeito abriu portas para outros tipos de estudos que foram fundamentais para muitas conceitos e tecnologias utilizadas direta ou indiretamente hoje.

2.1.2 Emissão termoiônica

Geralmente os elétrons conseguem se mover apenas dentro do metal quando excitados, porém não conseguem escapar. Quando um metal é aquecido o suficiente e os átomos absorvem essa energia térmica, alguns dos elétrons desse metal adquirem energia suficiente para se desprenderem da superfície do metal e ficarem próximas a essa superfície. Esse desprendimento ou escape é conhecido como emissão termoiônica, e a nuvem elétrica gerada por esse escape é denominado “Efeito Edison” (CURRY; DOWDEY; MURRY, 1990).

O efeito de emissão termoiônica foi muito importante para muitos estudos de física moderna, assim como estudos em dispositivos eletrônicos, e é tão essencial para

o funcionamento da máquina de raio X quanto o efeito fotoelétrico (CURRY; DOWDEY; MURRY, 1990).

Em 1853 Becquerel observou pela primeira vez o fenômeno, porém, somente no final do século dezenove Thomas Edison teve interesse pelo assunto em seus trabalhos com lâmpadas incandescentes (PAXTON, 2013). Edison notou que durante grandes períodos de funcionamento da lâmpada um resíduo escuro aparecia no interior do vidro da lâmpada. Também notou que sempre aparecia uma faixa branca no plano do filamento (PAXTON, 2013).

Após anos sem conseguir entender a causa da faixa branca, Edison posicionou outro eletrodo adjacente ao filamento e, com isso, observou fluxo de corrente elétrica no mesmo concluindo, portanto, que a emissão termoiônica de Becquerel era responsável por isso. Essa observação de Edison levou à sua primeira patente no que hoje chama-se diodo termoiônico (PAXTON, 2013).

Edison, entretanto, ainda não sabia explicar a origem dessa corrente. Após o abandono do estudo dessa anormalidade observada, Ambrose Flemming, conselheiro chefe de ciência da *Edison Eletric Light Co. of England* em 1890, continuou a pesquisa dos diodos termoiônicos em seu trabalho entitulado *On Electric Discharge between Electrodes at Different Temperatures in Air and in High Vacua* (FLEMING, 1890). Então em 1905 Flemming inventou a válvula termiônica ou diodo de tubo de vácuo.

Hoje entende-se que os elétrons evidentemente absorvem energia da agitação térmica das partículas de metal e, ao atingir uma certa energia mínima, os elétrons conseguem escapar de suas órbitas (BEISER, 2003). Para cada tipo de superfície, essa energia mínima pode ser determinada e sempre é próxima da função trabalho do efeito fotoelétrico discutido na Seção 2.1.1. De acordo com (BEISER, 2003), na emissão fotoelétrica os fótons de luz fornecem energia para que o elétron escape, enquanto na emissão termoiônica a temperatura tem esse papel.

2.2 FÍSICA DOS RAIOS X

Wilhelm Roentgen em 1895 descobriu uma radiação altamente penetrável de natureza até então desconhecida que ele chamou de raio X. Foi descoberto que o raio X é altamente penetrável em diversos tipos de matérias, viajam em linhas retas, não são afetados por campos elétricos ou magnéticos, passam por materiais opacos e causam a luminescência de substâncias fosforescente. Quanto mais rápidos forem os elétrons, mais penetrantes são os raios X e quanto maior a quantidade de elétrons, mais intenso é o feixe de raio X (BEISER, 2003).

Os raios X são ondas eletromagnéticas e radiação eletromagnética consiste em fótons e, de acordo com (SUETENS, 2009), a energia E de um fóton com frequência f e comprimento de onda λ é dado por

$$E = hf = \frac{hc}{\lambda} \tag{4}$$

onde h é a constante de Plank e c é a velocidade da luz no vácuo. Também $hc = 1.2397 \times 10^{-6} eVm$. O comprimento de onda dos raios X são na ordem de Angstrøms ($10^{-10}m$) e, por consequência, a energia dos fótons é na ordem de keV onde $1eV = 1.602 \times 10^{-19} J$. (SUETENS, 2009). Quando o raio X é gerado artificialmente, a energia se apresenta em dois diferentes processos: *bremsstrahlung* e radiação característica.

2.2.1 *Bremsstrahlung*

Quando um elétron, que é carregado negativamente, passa próximo a um núcleo carregado positivamente, ele é desviado de sua trajetória original ao ser atraído por esse núcleo. Então o elétron pode perder energia ou não. Caso o elétron não perca energia, esse fenômeno é chamado de dispersão elástica e nesse caso não será produzido raio-X. Entretanto se o elétron perder energia, ocorre uma dispersão inelástica e essa energia perdida é emitida na forma de um fóton de raio-X, essa radiação é chamada de *Bremsstrahlung* ou radiação branca (SHUNG; SMITH; TSUI, 2012).

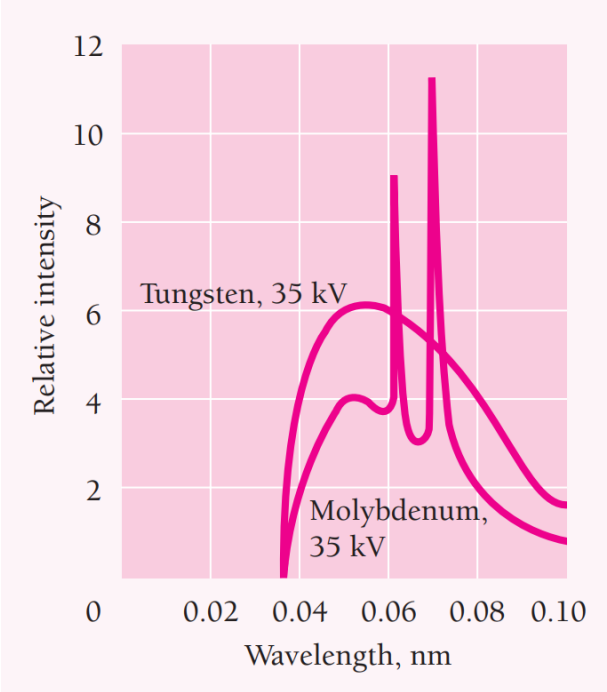
Essa probabilidade de o elétron perder ou não energia ao desviar a trajetória é dada pelo número atômico, quanto maior o número atômico, maior é a probabilidade de o elétron perder energia e emitir na forma de raios-X, ou seja, mais energético é o *Bremsstrahlung* (BEISER, 2003). Essa emissão de radiação varia de acordo também com a quantidade de núcleos com os quais o elétron pode interagir antes de parar e também a energia que ele carrega. Então a energia dos fótons de raios-X geradas nesse processo é distribuída em um amplo espectro de energia (SHUNG; SMITH; TSUI, 2012), de acordo com a Figura 4.

2.2.2 Radiação característica

Como pode ser observado na Figura 4, em certos comprimentos de onda ocorrem picos de energia. Esses picos de energia estão relacionados com uma probabilidade substancialmente maior de desaceleração e ressonância energética do material do ânodo baseado na sua configuração e estrutura atômica (NAJARIAN; SPLINTER, 2012). Isso acontece quando os elétrons que estão sendo bombardeados interagem com as camadas eletrônicas mais internas dos átomos (SHUNG; SMITH; TSUI, 2012) e desprendem um elétron da camada K.

Esse elétron desprendido da camada K poderia subir de nível atômico para preencher um espaço vago em uma das camadas superiores, porém a energia requerida para preencher esse espaço ou remover o elétron totalmente é praticamente a

Figura 4 – Espectro Raios-X do Tungstênio e Molibdênio



Fonte: Beiser (2003, p.70)

mesma (BEISER, 2003). Então o átomo com um elétron a menos na camada K acumula uma energia de excitação causada pela instabilidade eletrônica. Essa instabilidade força um elétron da camada superior a descer e completar a camada K, nesse momento essa energia acumulada pelo átomo é perdida em forma de fótons de raios-X (BEISER, 2003). Esse fenômeno é chamado de radiação característica.

2.3 GERAÇÃO DE RAIOS X

A geração dos raios-X ainda é feita de maneira muito semelhante à descoberta acidentalmente realizada por Wilhelm Röntgen (NAJARIAN; SPLINTER, 2012). Para a geração de raios-X, é necessário um tubo com dois eletrodos selados à vácuo, pois a presença de um gás pode variar o número de elétrons e reduzir a velocidade (SHUNG; SMITH; TSUI, 2012). Além do tubo, também é necessário um gerador elétrico para fornecer a energia necessária para alcançar uma condição favorável para a geração de raios-X quando aplicada ao tubo.

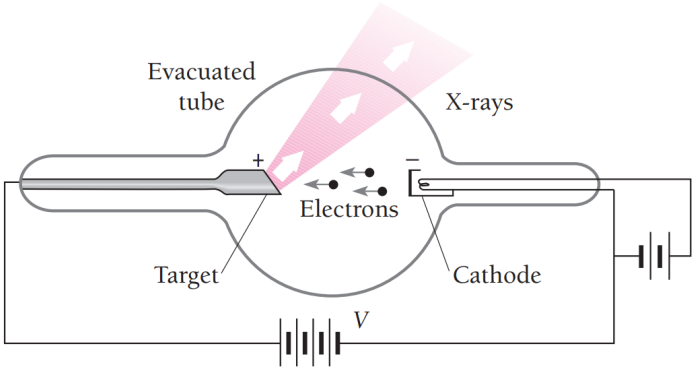
2.3.1 Emissão no tubo de raios-X

O tubo de raios-X é composto pelo cátodo, ânodo e uma ampola de encapsulamento. O cátodo é formado por dois elementos: o filamento de tungstênio e uma cúpula metálica para direcionar os elétrons emitidos pelo filamento. O filamento é feito geralmente de um fio tungstênio de aproximadamente $0,2mm$ de diâmetro e $1cm$

de comprimento em formato espiral (SHUNG; SMITH; TSUI, 2012).

Na Figura 5 é possível ver a exemplificação de um tubo de raios-X. Os elétrons são fornecidos por efeito termiônico no cátodo e são acelerados em direção ao ânodo por diferença de potencial elétrico (BEISER, 2003).

Figura 5 – Tubo de raios-X



Fonte: Beiser (2003, p.69)

Ao prover uma corrente elétrica ao fino filamento de tungstênio, o mesmo aquece. Quando um metal é aquecido seus átomos absorvem energia térmica e alguns dos elétrons no metal conseguem atingir um nível energético suficiente para se desprenderem da superfície (CURRY; DOWDEY; MURRY, 1990), através do efeito termoiônico discutido na Subseção 2.1.2.

Esses elétrons que escaparam do metal formam uma nuvem de elétrons ao redor do filamento que é conhecido como “Carga espacial” (SHUNG; SMITH; TSUI, 2012). Essa mesma nuvem de elétrons previne os outros elétrons do filamento de se desprenderem da superfície até que tenham energia térmica o suficiente para sobrepor a energia causada por essa carga espacial (CURRY; DOWDEY; MURRY, 1990). Essa tendência de limitar o desprendimento dos elétrons do metal é chamado de “efeito de carga espacial”.

Quando ocorre o desprendimento de alguns elétrons da superfície, a falta desses elétrons torna o filamento positivamente carregado. Isso tem como consequência a volta de alguns elétrons da carga espacial para o filamento novamente. Ou seja, a partir do momento em que filamento atinge uma temperatura suficiente para acontecer o efeito termiônico, ocorre rapidamente um equilíbrio em que o número de elétrons em êxodo do filamento é igual ao número de elétrons voltando ao filamento. Como consequência o número de elétrons presentes na carga espacial permanece constante e esse número depende da temperatura do filamento (CURRY; DOWDEY; MURRY, 1990).

Apesar de não ser o metal mais eficiente para o efeito termoiônico (CURRY; DOWDEY; MURRY, 1990), o Tungstênio é desejado por seu alto ponto de fusão ($3370°$C)

e pouca tendência de vaporizar. Além disso, ele possui um alto número atômico e é quimicamente estável (SOARES, 2008). Esse é um dos motivos pelo qual o material do ânodo também geralmente é o Tungstênio. Os ânodos para mamografia utilizam Molibdênio devido ao número atômico intermediário, portanto produz fótons de energia menores que são adequados à baixa densidade do tecido mamário.

Como mencionado anteriormente, os elétrons agora disponíveis na carga espacial ao redor da superfície do filamento são acelerados em direção ao ânodo por diferença de potencial. A maioria dos elétrons que chegam ao ânodo sofrem colisões e suas energias são transformadas em calor - apenas 1% de toda energia gerada é convertida em raios-X (SHUNG; SMITH; TSUI, 2012).

Quando muitos elétrons acelerados atingem o ânodo e alguns realizam o fenômeno de Bremsstrahlung, esses últimos liberam energia com uma frequência muito alta, em espectro de raios-X. Isso representa o efeito fotoelétrico inverso, ou seja, ao invés de um fóton carregado ser transformado em um elétron com energia cinética, um elétron com uma certa energia cinética (da aceleração) é transformado em um fóton com alta energia hv, discutida em 2.1.1 (BEISER, 2003).

De acordo com (CURRY; DOWDEY; MURRY, 1990):

> A unidade da corrente elétrica é o ampère, que pode ser definida como a taxa de "fluxo" quando 1 Coulomb de eletricidade flui através de um condutor em 1 segundo. O Coulomb é o equivalente à quantidade de carga elétrica carregado por $6,25 \times 10^{18}$ elétrons. Portanto, uma corrente no tubo de raios-X de $100mA$ $(0,1A)$ pode ser considerado como o "fluxo" de '$6,25 \times 10^{17}$ elétrons do cátodo para o ânodo em 1 segundo. A corrente de elétrons em um tubo de raios-X está em uma direção apenas (sempre cátodo para ânodo) ((CURRY; DOWDEY; MURRY, 1990), 1990, p.12).

O número de fótons de raios-X produzidos depende do número de elétrons atingindo o ânodo, ou seja, depende da corrente no tubo (dada em mA) que pode ser considerada como o fluxo de elétrons do cátodo para o ânodo em 1 segundo, e foi descoberto que essa relação é linear (SHUNG; SMITH; TSUI, 2012). Portanto, a exposição de raios-X produzidas depende da quantidade de elétrons por segundo que estão atingindo o ânodo multiplicado pelo tempo de ativação da diferença de potencial (dado em segundos). Ou seja, a exposição pode ser definida como $mA \times segundos$ ou simplesmente mAs.

Quando há fluxo de elétrons percorrendo o tubo, a força de repulsão mútua e a grande quantidade de elétrons faz com que haja uma indesejável dispersão dos raios ao atingir o ânodo, isso torna a imagem final menos nítida. Para evitar esse espalhamento, uma estrutura mecânica acoplada ao cátodo focaliza os elétrons emitidos do filamento.

Tubos de raios-X mais modernos contém dois filamentos dentro da cúpula focalizadora. Cada um dos filamentos é um fio helicoidal e eles são montados lado a lado ou até um acima do outro. Um desses filamentos é maior (CURRY; DOWDEY;

MURRY, 1990).

Apenas um dos filamentos é utilizado por vez. O filamento maior é geralmente utilizado para exposições mais longas, enquanto o menor é para exposições mais curtas porém com mais elétrons.

Então há dois tipos de foco possíveis, foco grosso (filamento maior) e foco fino (filamento menor). A seleção é feita escolhendo qual filamento será utilizado no exame, de maior ou menor comprimento.

2.3.2 Gerador de raios-X

Um gerador de raios-X é uma máquina que fornece energia elétrica para o tubo de raios-X. O gerador de raios-X não é exatamente um gerador elétrico, pois esse fornece energia elétrica a partir da conversão de energia mecânica (CURRY; DOWDEY; MURRY, 1990).

O gerador de raios-X adapta a energia elétrica disponível na rede em energia elétrica útil para a geração de raios-X . Essa energia fornecida pelo gerador precisa atingir dois objetivos quanto ao tubo de raios-X: aquecer o filamento e acelerar os elétrons do cátodo ao ânodo (CURRY; DOWDEY; MURRY, 1990).

Para tal, o gerador possui algumas malhas de controle e circuitos especializados para fornecer a energia adequada para os terminais do tubo, para o filamento e para ajustar o tempo de exposição correto do exame. O mecanismo dividido em duas partes: o painel de controle e o transformador. O painel de controle permite que o operador controle os parâmetros apropriados de kV, mA e tempo de exposição para um exame (CURRY; DOWDEY; MURRY, 1990).

Um botão de dois estágios é utilizado para dar início ao exame. No primeiro estágio o filamento é pré-aquecido e o disco do ânodo giratório é movimentado para velocidade final, seguindo uma curva ascendente de arranque. No segundo estágio do botão informa à malha de controle para iniciar a exposição (CURRY; DOWDEY; MURRY, 1990).

O segundo componente é o transformador, o qual possui um transformador de alta tensão para os terminais do tubo, um transformador de baixa tensão para o filamento e um grupo de retificadores para o circuito de alta (CURRY; DOWDEY; MURRY, 1990). As diferenças de potenciais no circuito podem atingir 150 mil Volts, e por esse motivo, o transformador é imerso em óleo, para que o óleo atue como isolante entre os componentes.

2.3.3 Grandezas ajustáveis na geração

Como discutido anteriormente, é necessário controlar a diferença de potencial entre os terminais do tubo (entre cátodo e ânodo). Esse potencial que é aplicado é

a energia que o elétron terá ao atingir o ânodo; a consequência disso é que o fóton possui uma energia limitada à energia do elétron que o criou (SOARES, 2008). Por conseguinte, a energia máxima de um feixe de raios-X, dada em kiloeletron-volts, é atrelada à diferença de potencial aplicada ao tubo (kVp). Logo, essa tensão é uma das grandezas que pode se ajustar em um disparo de raios-X.

Como discutido também em 2.3, o cátodo é aquecido por meio de uma corrente elétrica proveniente de uma fonte de baixa tensão e a saída dessa fonte é controlada por uma chave seletora de mA, que ao ser aumentada, aumenta também a corrente de filamento, tendo como consequência um maior número de elétrons desprendidos do filamento por efeito termiônico. Com esse aumento do número de elétrons desprendidos, por consequência, é aumentado o número de elétrons acelerados em direção ao ânodo, por tanto mais raios-X são gerados (CURRY; DOWDEY; MURRY, 1990).

Os elétrons que saem do filamento têm baixa velocidade porém, ao serem acelerados em direção ao ânodo, adquirem mais velocidade e como sempre há elétrons se desprendendo do filamento por efeitos discutidos anteriormente 2.3.1, então há uma corrente elétrica no tubo. Pode-se concluir, então, que há duas correntes elétricas presentes em partes distintas do tubo: a corrente de filamento na ordem de alguns Àmperes (SHUNG; SMITH; TSUI, 2012) e a corrente de tubo na ordem de mA.

O último parâmetro em um disparo de raios-X é o tempo de exposição que é o tempo no qual o circuito de geração fica ativo, emitindo raios-X em direção ao ânodo. Os temporizadores e botões de controles são ajustados pelo radiologista previamente antes do exame e esse ajuste de exposição leva em conta fatores como tamanho do paciente, kV e mA, de acordo com uma tabela técnica (SOARES, 2008).

Enquanto corrente elétrica no tubo (mA) é a quantidade de elétrons que partem do ânodo para o cátodo a cada segundo, o parâmetro mAs refere-se ao produto de mA $\times$ segundo, que representa o número total de elétrons que atingem o ânodo durante o tempo de exposição (SOARES, 2008).

O operador não seleciona o tempo de exposição, contudo, informa ao gerador para produzir uma quantidade de mAs com uma quantidade de kV. O gerador então tem o papel de modificar a corrente de tubo mA a cada período de tempo até atingir o valor de mAs escolhido. Logo é possível substituir uma corrente constante durante determinado tempo de exposição por um valor de mA variável com uma fração desse tempo de exposição, ou seja, o tempo de exposição do paciente ao raios-X diminui porém o número total de elétrons que atingem o ânodo é o mesmo (CURRY; DOWDEY; MURRY, 1990).

2.4 EQUIPAMENTOS SIMILARES

Como o objetivo desse trabalho é realizar o controle dos parâmetros discutidos na Subseção 2.3.3, é importante o estudo de equipamentos similares que já realizam esse tipo de controle via *software*. Na Figura 6a abaixo, é possível contemplar o conjunto radiológico de multi-propósito. Esse equipamento possui um sistema integrado de parametrização e acionamento do gerador assim como processamento e pós processamento de imagem, em um único *software*.

Em conjuntos radiológicos mais antigos, objeto de estudo no decorrer deste trabalho, há uma placa externa de parametrização e acionamento elétrico dos módulos de controle do gerador de raios-X. Abordagem a qual este trabalho busca atualizar, concentrando a parametrização em um *software* no computador, mas mantendo o acionamento elétrico em um *hardware* anexado ao gerador, no caso, um microcontrolador. Na Figura 6b é mostrada a placa externa utilizada em conjuntos antigos. Pode-se notar a parametrização de kV, mAs e foco.

Figura 6 – Equipamentos Similares

(a) Conjunto Radiológico de multipropósito.

(b) Placa externa de parametrização

Fonte: (a) Suetens (2009, p.25); (b) Imagem do autor.

3 REQUISITOS DO SISTEMA

Nesse capítulo serão abordadas as especificações de requisitos do sistema em desenvolvimento. A Seção 3.1 descreve a ideia geral do projeto, a qual abrange as perspectivas do produto, classes de usuários, funções do sistema, ambiente de operação e *design* da interface.

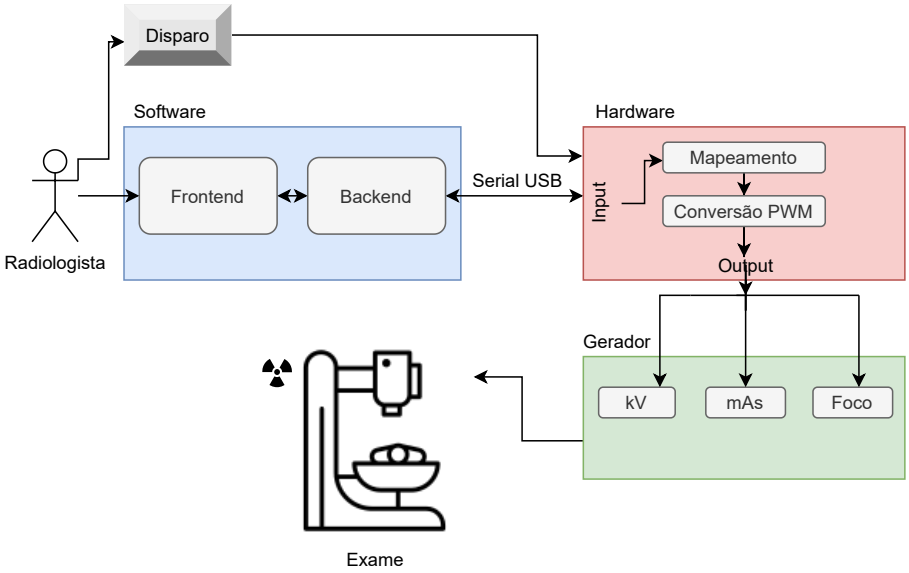
Logo após, a Seção 3.2 descreve os requisitos funcionais do sistema, ou seja, todas as características que precisam ser atingidas no final desse trabalho, tanto de *hardware* quanto de *software*.

Por fim, a Seção 3.3 aborda os requisitos não funcionais, ou seja, características adicionais que o sistema deve cumprir, isso envolve os requisitos de produto, organizacionais, externos e regras de negócio.

3.1 DESCRIÇÃO GERAL

Este projeto, a interface de controle de um equipamento de raios-X através de um computador, é uma proposta de atualização para o sistema radiológico estudado que atualmente utiliza apenas a placa de controle externa para ativação do gerador de raios-X. Esse desenvolvimento tem como finalidade centralizar as interações paramétricas com o operador em um único ambiente, abrindo a possibilidade de integração com outros sistemas presentes. Na Figura 7 pode-se contemplar o esquema geral do sistema.

Figura 7 – Diagrama resumido do sistema



Fonte: Elaboração própria.

3.1.1 Perspectiva de Produto

O desenvolvimento deste produto tem como perspectiva substituir por completo o atual conjunto de ativação do equipamento de raios-X digitalizado que hoje é por interface externa. O *software* deve se tornar parte do uso do operador no mesmo computador que executa os demais *software*s necessários para o processamento de imagens provenientes do disparo de raios-X. O *hardware* se tornará um módulo adicional obrigatório anexado ao gerador elétrico.

Esse produto, que envolve o conjunto *software* mais *hardware* intermediário e botão de disparo, fará parte do conjunto radiológico estudado para sistemas digitalizados.

3.1.2 Classes de Usuários e Características

O sistema terá duas classes de usuários primárias e uma classe de usuário secundária. Dentre as classes primária teremos o paciente e o técnico em radiologia. Na classe secundária teremos o administrador, podendo ser dividido entre administradores de manutenção e administradores hospitalares, como a gestão do hospital.

Para o paciente, quanto menor tempo de exposição melhor, então o projeto tem que englobar essa necessidade do cliente. Já para o comprador do produto ou administrador, o interessante é um *software* e *hardware* confiável, de fácil operação (menor custo de treinamento) e de baixa manutenção. Portanto o desenvolvimento do projeto também tem que seguir essas necessidades. Mais detalhes serão comentados em Requisitos Não Funcionais 3.3.

3.1.3 Funções do Produto

O produto deverá ter uma série de parâmetros configuráveis para que o técnico possa ajustar a fim de ter como resultado uma imagem de alta qualidade com a menor exposição de radiação. O produto deve ser capaz de configurar os seguintes parâmetros:

- Tensão entre ânodo e cátodo (kV);
- Corrente *versus* segundo do tubo (mAs);
- Seleção de foco (fino ou grosso).

Na Fase I do projeto são esses argumentos que devem ser configurados pelo usuário. Nas fases seguintes poderão ser adicionadas algumas funções como posição do tubo em relação ao detector, parâmetros de exposição automática (do inglês *Automatic Exposure Control* - AEC) e programa de partes do corpo entre outros. Entretanto, esse desenvolvimento será limitado a essas funções básicas de um programa de parametrização.

3.1.4 Ambiente de Operação

O *software* deverá ser executado em um sistema operacional compatível ao sistema operacional utilizado no *software* de diagnóstico de imagem. Atualmente esse *software* opera com Windows 10 H2. O *hardware* poderá ser anexado ao gerador que fica embaixo da mesa do paciente, lado a lado com as placas de controle do gerador, para facilitar a conexão do *hardware* proposto à placa do gerador. A comunicação entre o *software* e *hardware* inicialmente é interface de comunicação serial assíncrona do tipo USB. Porém, em projetos futuros, pode surgir novas formas de se comunicar com o *hardware*.

3.1.5 *Design*

A ideia inicial para a interface gráfica é desenvolver através do *framework* PyQt5 que é um conjunto de ferramentas de interface de usuário multiplataforma para a linguagem Python. Em termos de design, a interface precisa levar em conta a experiência do usuário (do inglês *User Experience* - UX) da forma mais simples e intuitiva possível, obrigatoriamente obedecendo aos requisitos de projetos propostos.

3.2 CARACTERÍSTICAS DO SISTEMA

Nessa seção serão discutidos os requisitos que o projeto precisa cumprir. Serão expostos os parâmetros e características de entrada e saída que deverão ser levados em consideração.

3.2.1 Descrição

O *software* precisa mapear os parâmetros de disparos inseridos pelo usuário e, em sequência, enviar para o *firmware* para que o *hardware* possa transmitir um nível de tensão analógico proporcional para os módulos de controle. Essa informação enviada do *software* para o *hardware* pode se dar de qualquer forma conveniente, desde que a informação de referência seja recebida corretamente pelo *hardware* e passada à placa de controle de acordo com uma tabela de referência. O esquemático do sistema pode ser contemplado na Figura 7.

3.2.2 Requisitos Funcionais

Os requisitos funcionais do sistema serão listados nessa seção e separados em duas partes: os requisitos de *software* e os requisitos de *hardware*. O sistema deve obrigatoriamente cumprir esses requisitos para que seja um sistema funcional.

São requisitos de ***software***:

- [RF01] Receber parâmetros de disparo do usuário: kV, mAs e foco;
- [RF02] Tecla e indicador de liga/desliga;
- [RF03] Enviar para o *hardware* as informações de disparo configuradas;
- [RF04] Monitorar os estados do sistema.

São requisitos de ***hardware***:

- [RF05] Receber os dados seriais do computador referente ao disparo;
- [RF06] Mapear os parâmetros de acordo com tabela de tensão de referência (Anexo A);
- [RF07] Enviar um nível de tensão em cada saída analógica (uma para cada parâmetro);
- [RF08] Possuir um botão de disparo de raios-X de dois estados.

3.3 REQUISITOS NÃO-FUNCIONAIS

Os requisitos não funcionais mostram como serão feitas as atividades descritas nos requisitos funcionais. Podemos dividir os requisitos em três partes: Requisitos de Produto Final, Requisitos Organizacionais, Requisitos Externo.

3.3.1 Requisitos de Produto Final

- [RNF01] Conceito *mobile first* na janela;
- [RNF02] Ambiente voltado para facilitar a experiência do usuário;
- [RNF03] Envio de informações isenta de erros;
- [RNF04] Garantia de limite de segurança dos parâmetros de disparo no *software* e *firmware*;
- [RNF05] Precisão de duas casas decimais na leitura de saída de tensão do *hardware* para todas as saídas analógicas;
- [RF06]Tolerância de +-3% V nas saídas analógicas.

3.3.2 Requisitos Organizacionais

- [RNF07] Sistema Operacional Windows compatível com *builds* do computador em vigor homologado (21H2);
- [RNF08] Processo de instalação e execução de *software*;
- [RNF09] Controle de versões;

3.3.3 Requisitos Externos

- [RNF10] Sem acesso à terceiros ao *software*;

- [RNF11] Informações pessoais dos usuários não podem ser vistas pelos operadores do sistema;
- [RNF12] Controle de acesso ao *software*.

3.3.4 Regras de Negócio

- [RNF13] Variação de kV entre 40 e 150, passos de 1 em 1 kV;
- [RNF14] Variação de mAs de 40 A 800;
- [RNF15] Se o usuário cancelar o exame, resetar os parâmetros;
- [RNF16] Se o usuário deslogar do sistema, reiniciar o mesmo;
- [RNF17] Se o usuário desligar o sistema, deslogar o usuário e fechar programa;

4 MATERIAIS E MÉTODOS

A interface de comando de um sistema radiológico pode ser modelada através de diagramas e abstrações para representar o sistema real e como ele funciona, para assim, facilitar o desenvolvimento do projeto. Além disso, com as representações orientadas a objetos podemos categorizar e unificar os dados, promovendo também a reutilização de código. Uma modelagem feita de maneira suficiente pode facilitar o entendimento do sistema como um todo e simplificar futuros desenvolvimentos para o sistema. O sistema contém quatro partes fundamentais além do próprio usuário, são elas:

- O ***software***, que promove a integração do usuário com o restante do sistema através da interface de usuário gráfica (GUI - do inglês, *Graphical User Interface*) e comunica com o microcontrolador Espressif ESP-WROOM-32D através de uma interface serial, utilizando o padrão de mensagem JSON (*JavaScript Object Notation*). O *software* é responsável por receber os dados do usuário, prepará-los para o envio e controlar a ordem das ações;
- O ***hardware*** externo, nesse caso um microcontrolador ESP32 que é responsável por receber os parâmetros do *software*, converter cada dado em um sinal de tensão correspondente e destinar essas tensões para os módulos de controle do gerador elétrico. Além disso, o *hardware* será responsável por liberar o funcionamento do botão de disparo de raios-X caso todos os parâmetros estejam completos, adequados e já enviados para o módulo de controle. Então o *hardware* comunica o *software* que ,por sua vez, comunica o usuário que já está liberado para disparo;
- Um **botão de acionamento** físico conectado ao microcontrolador que possua três estágios. O primeiro estágio é o estado em repouso do botão. O segundo estágio aciona o ânodo giratório e filamento do conjunto radiológico, controle que é feito pelo gerador elétrico. O terceiro estágio é o disparo do raio-X de fato. O acionamento do botão será delimitado pelo *hardware* conforme o item anterior.
- O **gerador elétrico** que é responsável por receber os dados de tensão vindos do *hardware* e utilizar as malhas de controle internas para preparar o tubo de raio-X para o exame proposto. No gerador elétrico também há uma entrada digital para efetuar o disparo do raio-X. Essa informação é recebida do *hardware*. Os sinais de tensão que o gerador precisa receber para cada parâmetro seguem uma tabela de referência definida no manual de operação do gerador.

A única ação que não é controlada pelo *software* é o disparo dos raios-X, que é feita pelo usuário através de um botão físico externo por demanda de normas.

O *Hardware*, porém, consegue delimitar o momento em que o botão de disparo pode ser considerado – se os parâmetros não foram inseridos pelo usuário totalmente, o disparo não pode ser feito. Nesse caso do botão, o *software* atuará apenas como um monitor do estado do botão, promovendo um retorno visual do que está acontecendo no sistema para o usuário.

O sistema será modelado através de diagramas UML (do inglês, *Unified Modeling Language*) em específico, quatro tipos: diagrama de Casos de Uso, diagrama de classes, diagrama de estados e diagrama de sequência.

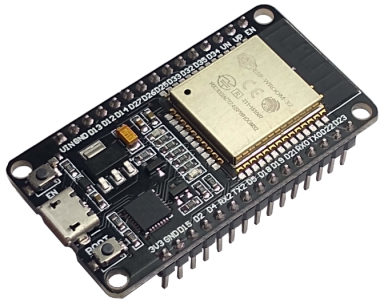
4.1 COMPONENTES UTILIZADOS

4.1.1 *Hardware*

O *hardware* principal selecionado foi o microcontrolador ESP-WROOM-32D da fabricante Espressif integrado em um módulo de desenvolvimento DOIT Esp32 DevKit V1, demonstrado na Figura 8. Esse microcontrolador possui dois núcleos de processamento (Xtensa® Dual-Core 32-bit LX6) com 448 KBytes de ROM e 520 Kbytes de RAM. Memória flash de 4MB e Clock de 80 à 240MHz ajustável. Além disso, possui WiFi 802.11 b/g/n: 2.4 à 2.5 GHz e Bluetooth BLE 4.2 BR/EDR e BLE (Bluetooth Low Energy). O microcontrolador ainda conta com GPIO com funções de PWM, I2C, SPI.

Para esse projeto, o microcontrolador foi escolhido por possuir modulação por comprimento de pulso (do inglês Pulse Width Modulation - PWM) e pela praticidade durante o desenvolvimento, oferecida pelo grande número de *frameworks* e bibliotecas disponíveis para utilização. Além disso, a escolha desse microcontrolador abre possibilidades para conexão sem fio com o computador do usuário.

Figura 8 – O módulo DOIT Esp32 DevKit V1.



Fonte: Imagem do autor.

Para o botão, foi escolhido um teclado de membrana do tipo matriz com dois botões (1x2), exibido na Figura 9 . Cada botão irá representar um estágio do botão

utilizado em uma aplicação real e esses botões, por sua vez, serão conectados ao microcontrolador.

Figura 9 – Teclado de membrana (1x2)

Fonte: Imagem do autor.

Para alimentação, durante o desenvolvimento, foi utilizado um módulo de fonte de alimentação 3.3V/5V, mostrado na Figura 10 que é uma placa desenvolvida para rápida instalação no protótipo, com entradas de alimentação por plug P4 ou USB Tipo-A. Possui pontos de contato de tensão ajustável de 3.3v e/ou 5v. Além disso, a placa possui um botão de liga e desliga, facilitando o processo de desenvolvimento.

Figura 10 – Fonte de alimentação 3.3V/5V

Fonte: Imagem do autor.

4.1.2 *Software* e *Firmware*

Para o desenvolvimento do *software* foi utilizada a linguagem Python em sua versão 3.8.6 em um Sistema Operacional Windows 10 21H2. Foram utilizadas algumas bibliotecas padrão inclusas na instalação do Python (*built-in*) como *json*, *os*, *time* e *logging* assim como outras dependências instaladas externamente utilizando *pip* 21.3.1 como instalador, como ,por exemplo, *pySerial 3.5, blessed 1.19.1*, *PyQ5 5.15.4* e *pyqt5-tools 5.15.4.3.2*.

Para o desenvolvimento do *firmware* foi utilizada a linguagem de programação

C++ junto com o framework “Arduino“ para a facilidade de implementações com algumas bibliotecas prontas. No *firmware* também foi utilizado uma biblioteca externa: ArduinoJSON (BLANCHON, 2021).

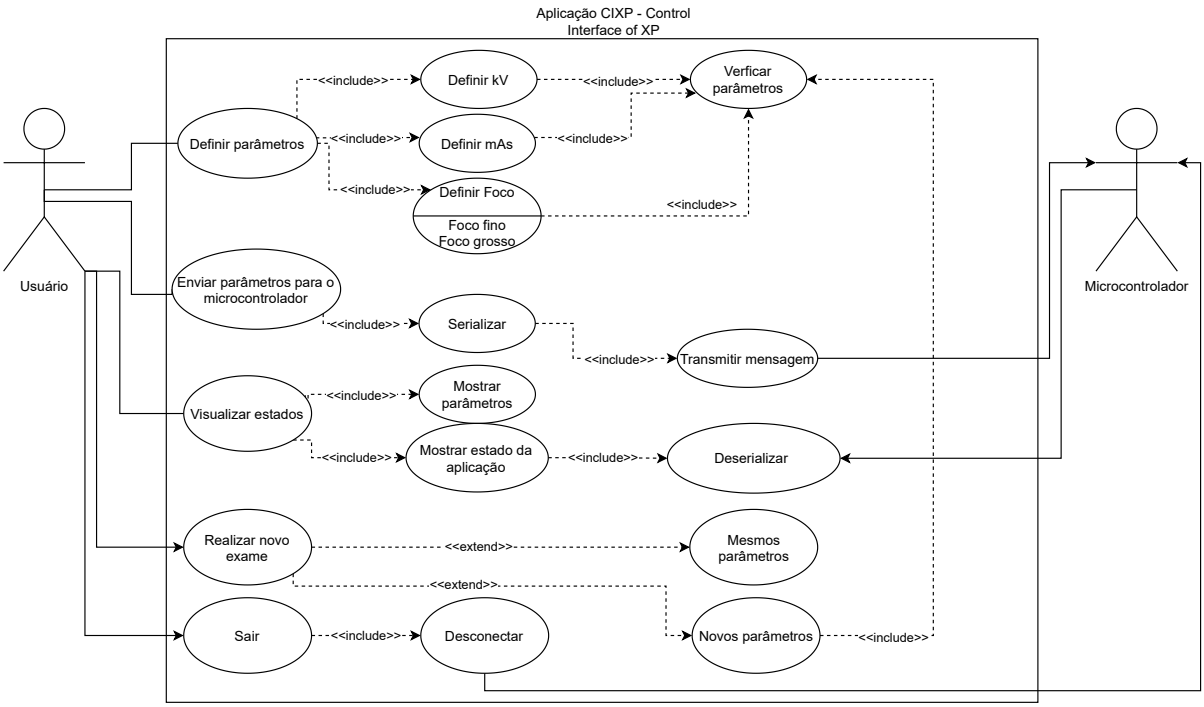
4.2 DIAGRAMAS DE CASO DE USO

O propósito do diagrama de casos de uso é demonstrar as diferentes maneiras de interação dos atores com o sistema. Nesse diagrama, o sistema foi dividido em duas partes: aplicação no computador que representa o *software* como um todo (Figura 11) e o microcontrolador que representa o sistema do *hardware* (Figura 12).

4.2.1 Diagrama de Casos de Uso da Aplicação

No diagrama da Figura 11 têm-se dois atores, o usuário e o microcontrolador. Ambos atores interagem com o sistema de forma distinta. O usuário pode definir os parâmetros do gerador, enviar esses parâmetros para o microcontrolador, visualizar o estado do sistema em cada estado em que ele se encontra. Além de outras ações como realizar um novo exame ou encerra a aplicação.

Figura 11 – Diagrama de Casos de Uso da aplicação



Fonte: Elaboração própria.

Quando o usuário define os parâmetros, implica na definição do kV, mAs, e foco (podendo escolher entre foco fino e foco grosso). Sempre que o usuário definir um certo parâmetro, este deve ser conferido para que esteja de acordo com os limites.

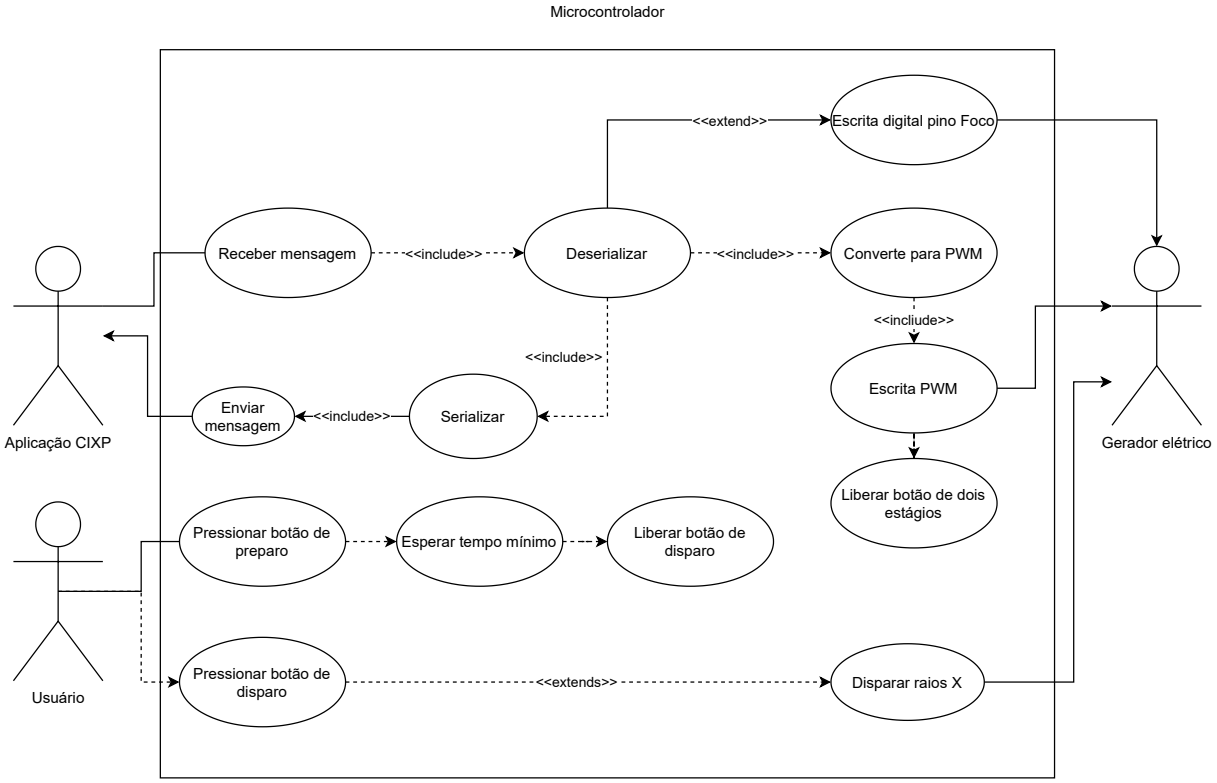
O usuário também pode enviar os parâmetros passados para aplicação para o microcontrolador. Esse caso de uso implica em serializar os parâmetros e transmitir para o microcontrolador. Além disso, o usuário pode visualizar o estado do sistema, para isso ele depende da comunicação entre a aplicação e o microcontrolador e os processos envolvidos durante o funcionamento do sistema. Essa atividade implica em mostrar os parâmetros e mostrar o estado da aplicação. Além de ver o estado do sistema, ele também poderá acompanhar para o exame atual os parâmetros que foram enviados.

Ao realizar um novo exame, o usuário pode escolher em realizá-lo com novos parâmetros ou manter os mesmos. Caso sejam novos parâmetros, os mesmos são verificados novamente, caso contrário, são apenas passados para o microcontrolador. Por fim, caso o usuário decida sair, essa atividade implica em desconectar e por conseguinte encerrar comunicação com o microcontrolador.

4.2.2 Diagrama de Casos de Uso do Microcontrolador

O segundo diagrama dos casos de uso é o diagrama do microcontrolador, que possui a aplicação *desktop* como um ator, além do botão do disparo e o gerador elétrico. Todos esses atores interagem com o sistema microcontrolador de alguma forma. A aplicação envia pacotes de mensagens para o microcontrolador, ou seja, o sistema tem uma ação que é a recepção dos dados. Isso implica em desserializar , converter em sinais de tensão de acordo com uma referência fixa e enviar através de portas analógicas para o gerador elétrico. Além disso, sempre que receber uma mensagem ele deve enviar uma resposta de recebimento.

Figura 12 – Diagrama de Casos de Uso do microcontrolador



Fonte: Elaboração própria.

O usuário possui duas ações sobre o microcontrolador (através do botão de dois estágios). A primeira é pressionar o botão de preparo, o qual inclui esperar o tempo mínimo com o botão ainda pressionado, para que o botão de disparo seja liberado. A segunda ação é o próprio botão de disparo que, se estiver liberado, pode ser pressionado por um tempo máximo e por consequência enviar um sinal de disparo de raios-X para o gerador de raios-X.

4.3 DIAGRAMA DE CLASSES

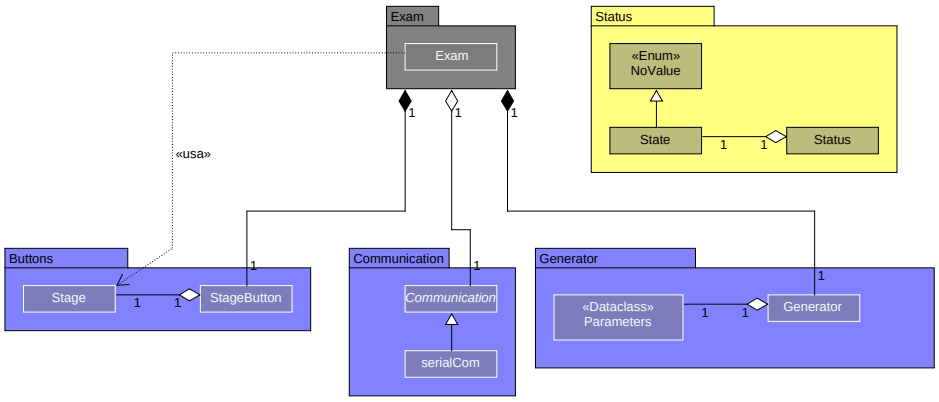
O diagrama de classe permite estruturar o sistema através da abstração utilizando classes, seus atributos e métodos, além de modelar os relacionamentos entre classes envolvidas para que então os processos, objetos do sistema e seus relacionamentos fiquem claros, facilitando a implementação e entendimento do sistema. Essa seção mostra a modelagem do desenvolvimento de *software* e *firmware* por meio dos diagramas de classes respectivos.

4.3.1 Diagrama de Classes de *software*

Na Figura 13 pode-se contemplar as classes do sistema de *software* modelado e seus relacionamentos. O diagrama possui as seguintes classes:

- Stage: Enumeração estágios do botão;
- StageButton: Representa botão de dois estágios;
- Communication: Abstração de comunicação entre o *software* e o *hardware*;
- serialCom: Responsável pela comunicação serial;
- Parameters: Representa os parâmetros do gerador;
- Generator: Representa o gerador elétrico;
- State: Enumeração dos estados da aplicação;
- Status: Representa o estado atual da aplicação;
- Exam: Classe contêiner que controla a funcionalidade do sistema vinculado à um exame com parâmetros bem definidos.

Figura 13 – Diagrama de Classes de *software*



Fonte: Elaboração própria.

4.3.1.1 *Relacionamentos das Classes de software*

O relacionamento da classe Exam com a classe StageButton é de composição, pois um objeto de StageButton só pode existir caso um Exam existir. Além disso, a classe Exam necessita de StageButton para seu funcionamento completo. A multiplicidade dessa classe é de 1 para 1.

A classe Communication possui uma relação de agregação com a classe Exam, que é uma relação mais fraca do que a de agregação quando trata-se de dependência. A classe Communication pode existir em outro contexto sozinha, entretanto, como a classe Exam é um contâiner, essa precisa de Communication para o funcionamento.

A classe Generator faz parte da classe Exam com um relacionamento de composição. Da mesma forma que StageButton, Generator só existe caso Exam exista. Nesse caso, a multiplicidade também é de 1 para 1, pois um exame possui um único gerador.

A classe Exam possui uma relação de dependência do tipo "uso"para com a classe Stage. Então, apesar da classe Exam usar a classe Stage, essa não possui outro tipo de dependência com Exam. A classe Stage tem uma relação de agregação com a

classe StageButton. A classe Parameters também possui uma relação de agregação com a classe Generator.

Como a classe Communication é uma classe abstrata, é natural que a classe serialCom possua uma relação de herança direta com a classe Communication. A classe State herda da classe NoValue apenas para ter acesso à classe Enum, a qual essa é herdada. Ainda, a classe State possui uma relação de agregação com multiplicidade 1 para 1 com a classe Status. Essa última é utilizada diretamente pela aplicação, assim como a classe Exam, e não têm outras relações com as demais classes.

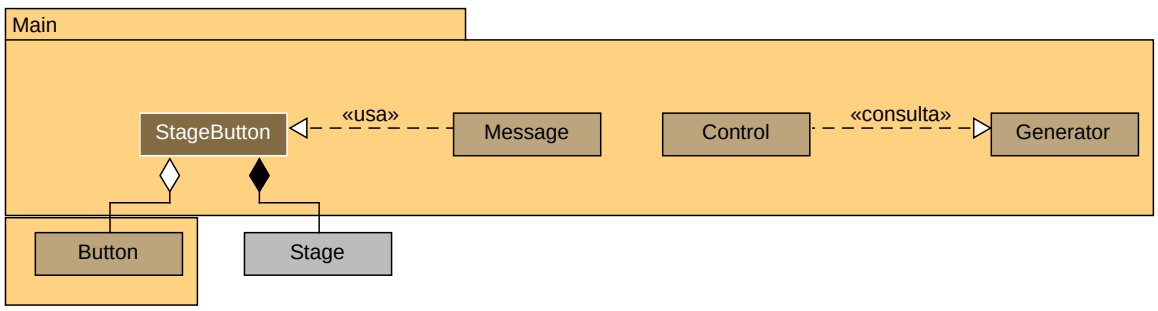
Ademais, as classes foram separadas de acordo com os módulos/arquivos nos quais estão inclusas. Então o diagrama de classes de *software* possui cinco módulos e organizadas de acordo com os relacionamentos.

4.3.2 Diagrama de Classes de *firmware*

Assim como para o *software*, para o desenvolvimento do *firmware* também foi criado um diagrama de classe. As classes contidas têm uma semelhança com as desenvolvidas pelo *software* por possuírem finalidades de representação semelhantes. Na Figura 14, é possível ver a modelagem das classes. As classes que fazem parte da modelagem do *firmware* são:

- Button: Representa um botão;
- Stage: Enumeração estágios do botão;
- StageButton: Representa botão de dois estágios;
- Message: Responsável pelo controle de mensagens;
- Control: Realiza o controle de saídas do *hardware*;
- Generator: Representa o gerador elétrico com parâmetros.

Figura 14 – Diagrama de Classes de *firmware*



Fonte: Elaboração própria.

4.3.2.1 Relacionamentos das Classes de firmware

Todas as classes são planejadas para compor o arquivo principal do *firmware* (main). A classe Button possui uma relação de agregação com a classe StageButton, sendo que StageButton contém um ou mais botões, porém um botão pode existir em um contexto separado.

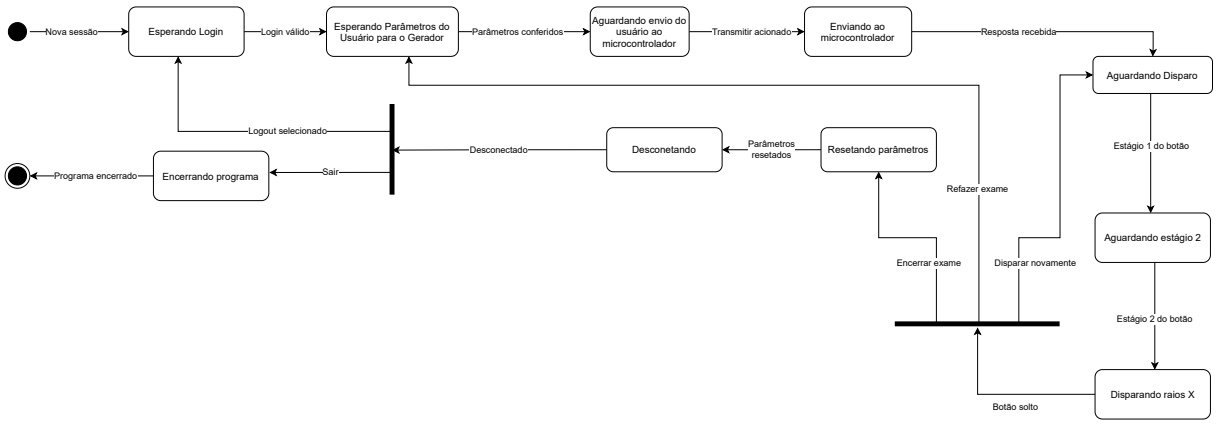
Já a classe Stage possui uma relação de composição com StageButton, isto é, Stage só existirá caso StageButton exista. Além disso, a classe StageButton também necessita de Stage para seu funcionamento correto, assim como em Button. Entretanto esta é uma relação mais forte de agregação.

A única relação que Message tem com outra classe é a de dependência com StageButton. A classe Message usa e depende de StageButton para seu funcionamento completo. Algo semelhante ocorre com a classe Control: A classe acessa/consulta a classe Generator. Ou seja, possui uma relação de dependência para o funcionamento correto.

4.4 DIAGRAMA DE ESTADOS

O diagrama da Figura 15 representa os estados possíveis do sistema do início ao fim do ciclo de vida, inclusive abrangendo um estado cíclico (no caso de o usuário querer realizar novamente o exame/disparo). Os estados e as transições estão indicadas no diagrama seguindo a ordem de realização do sistema. As guardas são condições vinculadas ao diagrama de classes.

Figura 15 – Diagrama de Estados



Fonte: Elaboração própria.

O sistema fica esperando o usuário realizar o *login* com sucesso. Quando o *login* está validado, o sistema fica em estado de espera pelos parâmetros do gerador. Quando os parâmetros são preenchidos, o sistema passa para o estado de aguardando envio ao microcontrolador. Assim que o usuário escolhe enviar, o sistema fica ocupado e envia a mensagem ao microcontrolador. Quando a mensagem de resposta for recebida

do microcontrolador, o sistema fica no estado aguardando disparo. Passa para o estado de aguardando estágio 2 quando recebe do microcontrolador um sinal de estágio 1. Quando o sistema recebe um sinal de estágio 2 pressionado, então ele muda para o estado de disparando raios-X. Quando o botão é solto, o sistema pode ir para três estados diferentes dependendo da ação do usuário, o qual pode escolher entre: encerrar exame, disparar novamente com os mesmos parâmetros ou escolher novos parâmetros para um novo disparo.

Caso seja escolhido disparar novamente, o sistema volta para esse estado (liberado/aguardando disparo). Caso o usuário queira editar os parâmetros para realizar um novo exame, o sistema volta para o estado de “Esperando parâmetros do usuário para o gerador”. Na escolha de *logout*, o sistema passa por uma série de estados de desconexão da máquina.

No momento em que o sistema está bloqueado e seguro, o usuário pode escolher entre encerrar o programa ao escolher sair - e então o programa vai para o estado final (encerramento) - ou apenas fazer o *logout* que o leva para o estado de esperando *login* novamente.

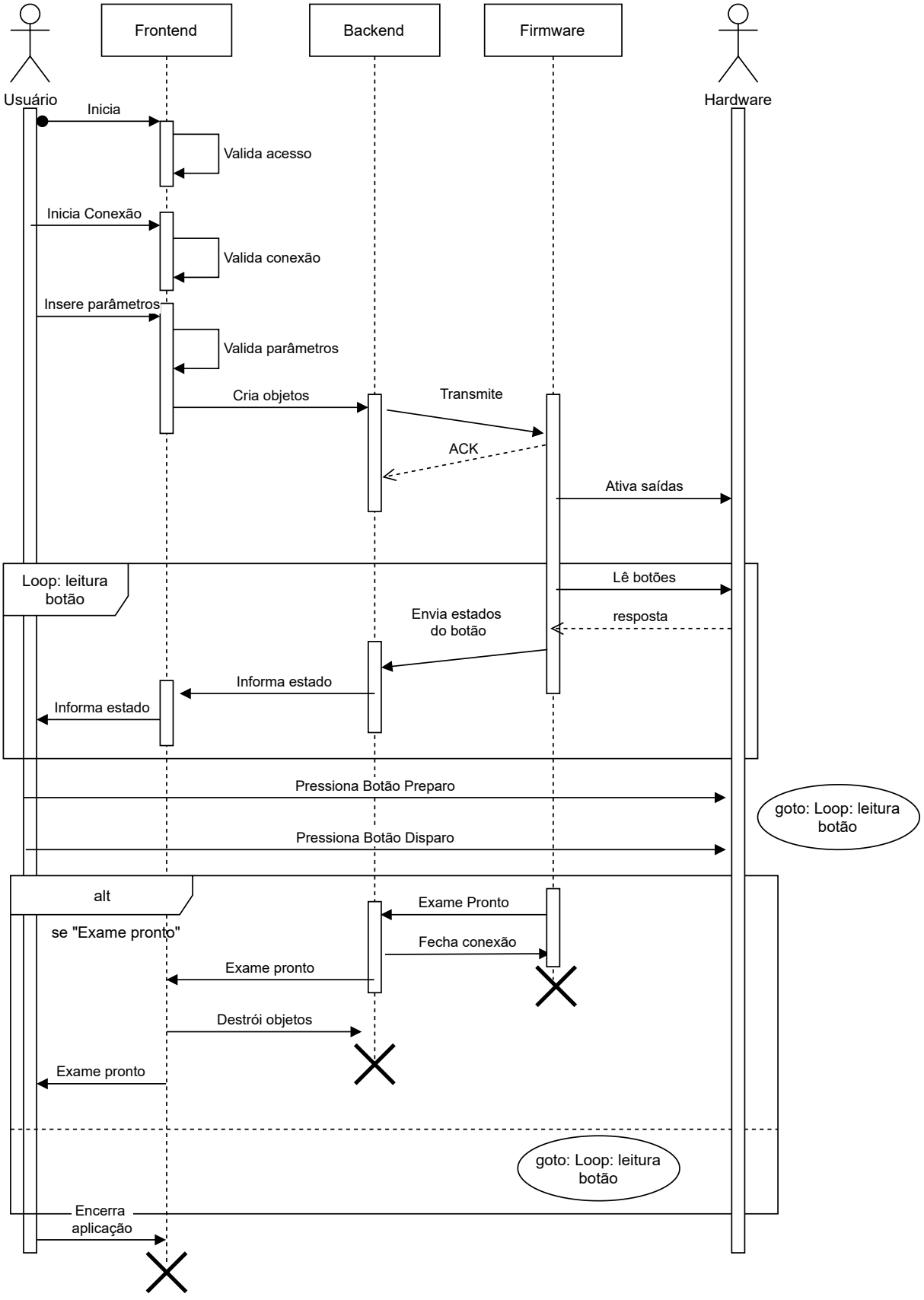
4.5 DIAGRAMA DE SEQUÊNCIA

Durante o processo há algumas etapas de verificação como a de *login* e verificação de parâmetros. Também há algumas mensagens de criação e destruição de objetos em determinados momentos.

O diagrama de sequência encapsula uma parte do processo que pode ser repetida, que é a parte de disparo de raios-X, que está reprensentada no *frame* em amarelo. O *hardware* foi representado por um outro ator para a simplificação do diagrama, esse ator indica a fronteira da aplicação, ou seja, a conexão com a comunicação serial do mundo externo.

O usuário inicia a interface gráfica e essa inicia as classes pertinentes (Exam, Parameters, Communication). A classe Exam cria um objeto de gerador e StageButton durante seu ciclo de vida, porque é um contêiner dessas classes de acordo com o Diagrama de classes (Seção 4.3).

Figura 16 – Diagrama de Sequência



Fonte: Elaboração própria.

5 DESENVOLVIMENTO

O desenvolvimento do projeto tem a necessidade de cumprir os requisitos de projeto propostos na Seção 3. Para tal, o desenvolvimento é dividido em quatro partes principais: *backend* de *software*, *firmware*, *frontend* de *software* e *hardware*. Todas as quatro partes em conjunto formam o desenvolvimento da interface de controle da máquina de raios-X, proposta nesse trabalho.

O desenvolvimento de códigos é dividido entre implementações de *software*, implementações de *firmware* e interface gráfica. No Apêndice C [7] é mostrada a árvore de arquivos do projeto.

O *backend* do *software* nesse projeto tem a função de ser o intermediário entre o *frontend* (interface gráfica) e o *firmware*. Enquanto isso o *firmware* tem a responsabilidade de realizar a interface entre o *software* e o gerador elétrico e seus módulos internos de controle. Já o *frontend* de *software* tem a finalidade de interagir com o usuário e passar as informações para o restante do *software* - por consequência *firmware* e gerador.

O desenvolvimento do *backend* e do *firmware* foram realizados em paralelo, uma vez que é preciso que ambos lados estejam comunicando entre objetos de representação semelhante. Por exemplo, para que uma classe que representa o botão esteja bem implementada no *software*, é preciso que no *firmware* também tenha uma classe semelhante para representar o mesmo objeto.

A montagem básica do *hardware* envolve conectar o módulo do ESP32 em uma porta USB para que seja possível implementar algumas funções iniciais e isso é bom no começo do desenvolvimento. Porém para o desenvolvimento completo é necessário conectar os demais componentes que também pode ser feita paralelamente ao desenvolvimento do *firmware*.

Nas próximas seções, serão discutidos os desenvolvimento do projeto e dos códigos envolvidos. No caso dos arquivos de implementação do *firmware*, as definições das classes (arquivos *header* .h) estão junto com a explicação nas próximas seções enquanto os arquivos de implementação estão no Apêndice B [7]. Já os arquivos em Python do *software* estão por completo na explicação.

5.1 *BACKEND* DE *SOFTWARE*

O *backend* é composto por módulos, classes e um programa principal agregando tudo isso em uma única execução. Nessa seção serão apresentadas com detalhes todas essas partes desenvolvidas. É importante lembrar que o *backend* é desenvolvido em Python no computador utilizando um editor de texto e um terminal do

sistema operacional.

5.1.1 Módulos e Classes

Com objetivo de estruturar o desenvolvimento voltado à programação orientada a objeto, foram desenvolvidas classes que representam partes do sistema. Essas classes têm objetivo de abstrair um conjunto de objetos com características similares. Na aplicação, as classes foram divididas em arquivos (ou módulos) para melhor organização e importação desses módulos, são eles: Generator, Communication, Buttons, Status e Exam. Nas próximas seções é descrito com detalhe o funcionamento e estrutura de cada classe dentro de cada módulo.

5.1.1.1 Módulo Generator

Esse módulo representa o gerador elétrico do conjunto radiológico e é composto por duas classes: Generator e Parameters. A relação entre essas classes é de agregação, ou seja, a classe Generator agrega Parameters.

```
1  #!/usr/bin/env python
2
3  from dataclasses import dataclass
4  import logging
5  import json
6
7  @dataclass(frozen = False, order=True)
8  class Parameters:
9      __parameters = {
10         "kV": 0,
11         "mAs": 0.0,
12         "focus": "fine",
13     }
14
15     def __init__(self, kV: int, mAs: float, focus: str) -> None:
16         try:
17             self.kV = kV
18             self.mAs = mAs
19             self.focus = focus
20         except Exception as e:
21             logging.exception("Parameters have incorrect types")
22             raise e
23
24     def dumps(self):
25         togo = {"parameters": self.__parameters }
```

```
26          package = json.dumps(togo)
27          logging.info(f"Dictionary with parameters: {package}")
28          return package
29
30      @property
31      def kV(self):
32          return self.__parameters["kV"]
33
34      @property
35      def mAs(self):
36          return self.__parameters["mAs"]
37
38      @property
39      def focus(self):
40          return self.__parameters["focus"]
41
42      @kV.setter
43      def kV(self, value):
44          if not (isinstance(value, int)):
45              logging.exception("Not an integer on field kV",
                    ↪ exc_info = False)
46              raise TypeError
47
48          if(value <= 150 and value >= 0):
49              self.__parameters["kV"] = value
50          else:
51              logging.exception("Please insert kV value between 0 and
                    ↪  150.", exc_info = False)
52              raise ValueError
53
54      @mAs.setter
55      def mAs(self, value):
56          if not (isinstance(value, float)):
57              logging.exception("Not an float on field mAs", exc_info
                    ↪  = False)
58              raise TypeError
59
60          if(value <= 800.0 and value >= 0.0):
61              self.__parameters["mAs"] = value
62          else:
```

```
63              logging.exception("Please insert mAs value between 0.0
                    ↪ and 800.0", exc_info = False)
64              raise ValueError
65
66      @focus.setter
67      def focus(self, value):
68          if not (isinstance(value, str)):
69              logging.exception("Not an string on field Focus")
70              raise TypeError
71
72          if(value == "thick" or value == "fine"):
73              self.__parameters["focus"] = value
74          else:
75              logging.exception("Please insert 'thick' or 'fine'")
76              raise ValueError
77
78  class Generator:
79      __parameters: Parameters
80
81      def __init__(self, parameters: Parameters) -> None:
82          self.__parameters = parameters
83
84      @property
85      def parameters(self):
86          return self.__parameters
87
88      @parameters.setter
89      def parameters(self, new_parameter: Parameters):
90          self.__parameters = new_parameter
```

5.1.1.1.1 Classe Parameters

A classe Parameters representa o tipo de dado, parâmetros de configuração do exame que serão enviados ao gerador. Então há três tipos de dados: kV, mAs, e foco, que foram discutidos na Subseção 2.3.3. Os tipos de dados são diferentes entre parâmetros e é necessário garantir que haja coerência dos valores inseridos dos atributos e a aplicação real. Os limites e tipos da dados foram definidos nos requisitos descritos na Seção 3.

O valor de kV é um número inteiro e ele deve estar compreendido entre 40 e 150 para condizer com os limites aceitos pelo gerador elétrico. Porém´, é necessário que o valor zero seja aceito para representar tensão nula no gerador, para que ,ao desligar

o gerador, seja enviado para o módulo de controle tensão zero. Para isso, a classe deve aceitar valores entre 0 e 150, inclusive valores entre 1 e 39. Porém o *firmware* realiza um filtro em que somente valores entre 40 e 150 são válidos enquanto que para qualquer outro valor seja colocado tensão zero na saída PWM do microcontrolador.

O valor atribuído para mAs precisa ser um número de ponto flutuante (*float*) e devem estar compreendidos entre 40.0 e 800.0, podendo aceitar valores menores [0.0, 40.0) que no *firmware* são tratados como zero, assim como descrito no parâmetros de kV. É importante garantir os valores estejam nos limites corretos e que o tipo de dado seja preservado para que corresponda a um valor coerente na saída PWM.

Para o foco é possível duas situações: fino e grosso. O tipo de dado pouco importa desde que seja acordado entre *software* e *firmware* a diferença no que representa o foco fino e foco grosso. O tipo de dado poderia ser booleano (verdadeiro ou falso), inteiro zero e diferente de zero, caractere representando fico fino ou foco grosso, ou então um vetor de caracteres (*string* ou palavra) representando os dois estados possíveis. Para facilidade de compreensão do código, foi escolhido diferenciar foco fino como a *string* "*fine*" e foco grosso como *string* "*thick*".

Dataclass é um tipo especial de classe de Python e é mais simples para implementar. É composto, basicamente, por dados (com identificadores e valores) e dataclass possui alguns métodos básicos já implementados. O objetivo é reduzir a repetição de implementações e também modularizar algumas tratativas e propriedade dos dados para que mediante qualquer mudança de adição de parâmetros, não tenha que ser reimplementado muitos métodos.

Nesse sentido, para o desenvolvimento da classe de Parameters é utilizado o conceito de dataclass de Python. Existe um único atributo dentro dessa classe, esse parâmetro é um dicionário com três chaves e seus valores. Cada uma das chaves corresponde a um parâmetro de exame. Esse formato foi escolhido para simplificar o envio de mensagens utilizando o formato JSON, que é um formato leve de troca de dados. Para o programador é simples de entender e utilizar, e para o computador é leve e rápido de analisar e processar.

A classe possui um construtor padrão que recebe como parâmetros kV, mAs, e foco. Para a atribuição dos valores recebidos às variáveis utiliza-se um decorador de propriedade. O *setter* desse decorador verifica o tipo de dado e os limites possíveis. Caso o dado seja inválido, ele levanta um sinal de erro para a camada superior da aplicação. A mesma estratégia de atribuição é feita para os três tipos de dados recebidos, cada um com sua peculiaridade de limite e tipo de dado.

Outro método dentro dessa classe é o *dumps()*, que realiza o empacotamento dos parâmetros em uma única *string* com um formato já esperado pelo *firmware*. Esse método utiliza o atributo *__parameters* interno da classe e cria um JSON com esse dicionário, converte em string utilizando *json.dumps()* da biblioteca *json*. O método,

portanto, não recebe nada e retorna uma *string* com o dicionário de parâmetros.

5.1.1.1.2 Classe Generator

A classe Generator é um contâiner que possui um atributo do tipo Parameters. Essa classe possui um construtor que recebe um objeto de Parameters e atribui à variável *__parameters* interna da classe. Além disso, ela possui métodos *getter* e *setter* utilizando um decorador de tipo *property*. Generators é uma classe enxuta porém é importante diferenciarmos o que é um parâmetro e o que é um gerador para que a representação fique fiel aos objetos reais, pois um gerador possui parâmetros e é utilizado em um exame, que será visto mais para frente na classe Exam. A classe Generator pode ser incrementada com atributos específicos para cada gerador, por exemplo modelo, nome e número de série.

5.1.1.2 Módulo Communication

O módulo Communication é responsável por representar toda a comunicação entre o *software* e o *firmware*. Tratativas de conexão, reconexão, envio e recebimento de mensagens é algo essencial nesse módulo, porém essas tratativas têm que ser independentes do tipo de conexão física (guiada ou não). Para tal, é criada uma classe Communication que é uma classe abstrata.

A classe abstrata é uma classe especial que não pode ser instanciada, apenas herdada. Ela serve para criar uma representação usando conceitos de herança de classe. Possui métodos abstratos e que não possuem implementação, apenas uma assinatura que deve ser implementada por cada subclasse. No caso de comunicação, toda comunicação possui algumas ações básicas, porém cada tipo de conexão possui uma peculiaridade em sua implementação que é a interface de comunicação em si.

Como a interface de comunicação nesse projeto é serial e é do tipo receptor e transmissor assíncrono universal (UART), então faz sentido ter uma classe de comunicação serial que implementa os métodos pertinentes à essa aplicação. Para tal, é criada uma classe serialCom que herda de Communication.

```
1 #!/usr/bin/env python
2
3 import serial
4 import logging
5 from time import sleep
6 from abc import ABC, abstractmethod
7
8 class Communication(ABC):
9     @abstractmethod
```

```
10      def connect(self):
11          raise NotImplemented
12      @abstractmethod
13      def reconnect(self):
14          raise NotImplemented
15      @abstractmethod
16      def send(self, data):
17          raise NotImplemented
18      @abstractmethod
19      def read(self):
20          raise NotImplemented
21      @abstractmethod
22      def close(self):
23          raise NotImplemented
24
25  class serialCom(Communication):
26      def __init__(self, serialPort = 'COM3', serialBaud = 115200,
            ↪ timeOut = 2.5, maxAttempts = 5):
27          super().__init__()
28          logging.info("Serial communication is created (not
                ↪ connected yet)")
29          self.port = serialPort
30          self.baud = serialBaud
31          self.timeout = timeOut
32          self.maxAtt = maxAttempts
33          self.com_id = "Handshake-ID:CIXP"
34          self.serialConnection = serial.Serial(None, self.baud,
                ↪ timeout=self.timeout,
35                  parity=serial.PARITY_NONE,bytesize=serial.EIGHTBITS
                        ↪ ,
36                  stopbits=serial.STOPBITS_ONE)
37          self.serialConnection.port = serialPort
38
39      def connect(self):
40          attempt = 0
41          logging.info('Connecting to: ' + str(self.port) + ' at ' +
                ↪ str(self.baud) + ' BAUD.')
42          while(attempt <= self.maxAtt):
43              try:
44                  self.serialConnection.open()
45                  logging.info('Connected to ' + str(self.port) + '
```

```
                    ↪ at ' + str(self.baud) + ' BAUD.')
46                  _trash = self.serialConnection.read()
47                  self.serialConnection.reset_input_buffer()
48                  self.serialConnection.reset_output_buffer()
49                  return True
50              except:
51                  logging.exception(f"(Attempt {attempt}): Failed to
                        ↪ connect with {str(self.port)} at {str(self.
                        ↪ baud)} BAUD.",exc_info=False)
52                  attempt += 1
53                  sleep(1)
54          return False
55
56      def reconnect(self):
57          logging.info('Reconnecting to ' + str(self.port) + ' at ' +
                ↪  str(self.baud) + ' BAUD.')
58          self.serialConnection.close()
59          if(self.connect() == False):
60              return False
61          else:
62              return True
63
64      def handshake(self):
65          attempt = 0
66
67          while(attempt <= self.maxAtt):
68              self.send("ID request")
69              incoming = self.read()
70              logging.info(f"Data received: {incoming}")
71              if(incoming == self.com_id):
72                  logging.info(f"Handshake ID is correct: {incoming}
                        ↪ == {self.com_id}")
73                  return True
74              else:
75                  attempt += 1
76                  logging.warning(f"Handshake ID is incorrect: {
                        ↪ incoming} != {self.com_id}. Trying again...")
77          return False
78
79      def send(self, data):
80          try:
```

```
81              if self.serialConnection.isOpen():
82                  logging.info(f"Sending data via serial: {data}")
83                  self.serialConnection.write(data.encode('ascii'))
84                  self.serialConnection.flush()
85                  logging.info("Data was sent")
86                  return True
87          except:
88              logging.exception(f"Fail to send. Device may be not
                    ↪ connected anymore.")
89              return False
90
91      def read(self):
92          try:
93              incoming = self.serialConnection.readline().decode("utf
                    ↪ -8")
94              incoming = incoming.replace('\r', '').replace('\n', '')
95              # logging.debug(f"Data received: {incoming}")
96              return incoming
97          except Exception as e:
98              logging.exception("Fail to read. Device may not be
                    ↪ connected anymore")
99              raise e
100
101     def close(self):
102         logging.info("Disconnecting...")
103         try:
104             self.serialConnection.close()
105             return True
106         except Exception as e:
107             logging.exception("Could not disconnect.")
108             raise e
```

5.1.1.2.1 Classe Communication

A classe Communication é uma classe abastrata, como dito anteriormente, e possui os métodos *connect()*, *reconnect()*, *send()*(), *read()*() e *close()*. Todas elas levantam uma exceção *NotImplemented* caso sejam usadas diretamente, pois o objetivo é que as classes que herdam ela implementem esses métodos.

5.1.1.2.2 Classe serialCom

A classe serialCom possui um construtor que recebe a porta serial conectada ao computador (padrão COM3), *baud rate* - a taxa de *bits* por transmissão de sinal (padrão 115200), *timeout* - o tempo limite de conexão (padrão 2.5), 'maxAttempts' - o máximo de tentativas para realizar a reconexão e o *handshake* (padrão 5). Além de atribuir as variáveis, o método cria o objeto de comunicação serial (da biblioteca *pyserial*) porém não inicia a conexão ainda.

O método *connect()* abre a comunicação e lê possíveis lixos de memória no *buffer* e descarta. Caso a conexão tenha tido sucesso, o método retorna 'True', caso contrário tenta novamente por 'maxAttempts' vezes. Já o método *reconnect()* fecha a comunicação e tenta conectar novamente com o método *connect()*, caso consiga retorna 'True', caso contrário retorna 'False'. É importante utilizar o método *reconnect()* e não apenas *connect()* para que a conexão seja de fato fechada antes de tentar novamente a conexão. Esses métodos podem ser chamados diretamente a partir da interface gráfica e interagir com o usuário.

O método *handshake*() envia uma string "ID Request" para o microcontrolador, o qual está esperando por essa mensagem e responde com a palavra "Handshake-ID:CIXP". Caso a palavra não seja exatamente igual ao ID esperado, o *handshake* falhou e deve ser tentado de novo por maxAttemps. A construção e uso de um *handshake* é necessário para garantir que não há problemas de comunicação após conectado. Então, enquanto a palavra requisitada não estiver correta, houve algum erro de comunicação ou o dispositivo não é o correto e, caso o erro persista, a operação deve ser cancelada.

Para fazer o envio das mensagens ao microcontrolador, é utilizado o método *send()*(data), o qual verifica se a comunicação está aberta, e se estiver, envia o dado recebido utilizando codificação ASCII. O envio é feito com o método de *pyserial write()*. Caso não seja possível enviar por algum motivo é retornado Falso, caso contrário, é retornado 'True'. O método *read()*() tenta ler uma linha decodificando em utf-8 e também substitui possíveis caracteres especiais de linha como '\n' ou '\r' por uma *string* vazia. No sucesso de leitura é retornado o valor lido do serial, na falha de leitura é levantada uma exceção. Quanto ao método *close()*, basicamente ele chama o método *close()* do pySerial e retorna 'True' caso sucesso.

5.1.1.3 Módulo Buttons

O módulo Buttons representa o botão de disparo físico e possui duas classes: StageButton e Stage. Como descrito na Seção 3, o *software* não tem nenhuma ação sobre o botão físico então toda a parte de *software* relacionado ao botão é apenas um visualizador dos estados que o *firmware* transmite para a aplicação.

A classe Stage é do tipo enumeração e possui três estados diferentes: IDLE, PREPARE e SHOOT. O objetivo da classe é apenas enumerar os possíveis estados do botão. O estado IDLE representa o botão em estado normal, não pressionado, porém pronto para disparo. Já o estado PREPARE indica que o primeiro estágio do botão foi pressionado e, nesse momento, a preparação do exame está sendo realizada (giro do ânodo giratório e pré aquecimento de filamento).

O estado SHOOT representa o segundo estágio do botão e que o disparo de raios-X está sendo efetuado. Esse comportamento continua por um tempo determinado de segurança, que está definido no *firmware* e será discutido na Seção Firmware 5.2. Após esse tempo, o *firmware* envia para o *software* uma mensagem de exame pronto (Done) e então o exame encerra.

A classe StageButton é um controlador que representa o estado atual do botão, para que então possa ser acessado em outras partes do código. Essa classe possui um construtor que apenas inicia o atributo privado *__stage* para o estado IDLE. Nessa classe também é implementado o decorador de propriedade para possuir o *get* e *set* desse atributo *__stage*.

```
1  #!/usr/bin/env python
2  from enum import Enum
3
4  class Stage(Enum):
5      IDLE    = 0
6      PREPARE = 1
7      SHOOT   = 2
8
9  class StageButton:
10     __stage: Stage
11
12     def __init__(self) -> None:
13         self.__stage = Stage.IDLE
14
15     @property
16     def stage(self):
17         return self.__stage
18
19     @stage.setter
20     def stage(self, new_stage: Stage):
21         self.__stage = new_stage
```

5.1.1.4 Módulo Status

O módulo Status tem a finalidade de representar os possíveis estados da aplicação/*software*. Ele é composto por três classes: NoValue, State e Status. A classe NoValue é apenas uma classe herdada de Enum e que faz uma sobrecarga do método *__repr__* para formatar a representação de *enum* sem que seja preciso um valor inteiro em sua representação.

A classe State é herdada de NoValue e tem o objetivo de enumerar os possíveis estados do sistema. Esses estados podem ser chamados pela aplicação em qualquer momento tanto para atribuir quanto para monitorar o estado da aplicação e mostrar para o usuário. Essa enumeração especial está descrita na Tabela 1:

Tabela 1 – Possíveis estados da aplicação.

Estado	Descrição
STARTING	Iniciando conexão
STAND_BY	Esperando parâmetros
WAITING	Esperando o envio
BUSY	Ocupado enviando
READY	Parâmetros enviados
PREPARING	Ânodo e filamento sendo preparados
SHOOTING	Disparando raios-X
SHUTTING_DOWN	Encerrando aplicação
OFFLINE	Aplicação está offline

Fonte: Elaboração própria.

A classe Status é composta por um atributo (*__state*) tipo State e possui um construtor e decoradores de propriedade para que esse atributo possa ser usado por outras funções e classes que acessam o *setter* e *getter* como se estivessem usando um atributo público porém, ele é privado, acessado e atribuído de maneira específica para proteger o tipo de dado. Enquanto a classe State apenas enumera os estados, a classe Status utiliza esses estados para armazenar (e mudar quando preciso) o estado da aplicação, ou seja, a classe Status apenas utiliza a classe State, mantendo-a externa para que outras partes do código possam usá-la sem necessariamente acessá-la através de um objeto de Status.

O método *__init__* de Status apenas inicia o atributo *__state* com o padrão State::STARTING. O método status (com decorador de propriedade) apenas retorna o estado, que pode ser consultado através de seu método __repr__ previamente comentado. Por exemplo, em um determinado momento é necessário saber qual estado em que a aplicação se encontra, então basta acessar como *objeto.status.name* para acessar o nome do estado ou então *status.status.value* para acessar a descrição. Para facilitar o acesso à descrição, foi criado um método (*description()*) que retorna a descrição do estado, portanto, pode-se acessar com *objeto.description()*.

```
#!/usr/bin/env python

import logging
from enum import Enum
from turtle import done

class NoValue(Enum):
    def __repr__(self):
        return '<%s.%s>' % (self.__class__.__name__, self.name)

class State(NoValue):
    STARTING      = "Iniciando conexao"
    STAND_BY      = "Esperando parametros"
    WAITING       = "Esperando o envio"
    BUSY          = "Ocupado enviando"
    READY         = "Parametros enviados"
    PREPARING     = "Anodo e filamento preparados"
    SHOOTING      = "Raios-X disparados"
    DONE          = "Exame finalizado"
    SHUTTING_DOWN = "Encerrando aplicacao"
    OFFLINE       = "Aplicacao esta offline"

class Status():
    __status: State

    def __init__(self) -> None:
        self.__status = State.STARTING

    @property
    def status(self):
        return self.__status

    @status.setter
    def status(self, new_state: State):
        if not (isinstance(new_state, State)):
            logging.exception("Not an instance of State class",
                ↪ exc_info = False)
            raise TypeError
        self.__status = new_state
```

5.1.1.5 *Módulo Exam*

O módulo Exam é um contâiner de classes e serve para realizar os procedimentos relacionados ao exame com uma ordem determinada, além de encapsular algumas funções. O exame possui como atributos um objeto do tipo Generator, um do tipo Communication e outro do tipo StageButton. Esses atributos fazem sentido serem parte da classe Exam, pois um exame precisa desses objetos para ser realizado. A classe Exam possui uma relação de composição com Generator e com StageButton, pois ambos só podem existir caso exista um exame. Já a classe Communication possui uma relação de agregação com a classe Exam, pois Exam necessita de Communication para realizar sua atividade enquanto Communication não depende de Exam, ou seja, pode existir sozinha em outros contextos.

```python
#!/usr/bin/env python

from .generator import Parameters, Generator
from .communication import Communication
from .buttons import StageButton, Stage

from time import sleep
import logging
import json

class Exam:
    __generator: Generator
    __com: Communication
    __stagebutton: StageButton

    def __init__(self, exam_parameters: Parameters, communication:
        ↪ Communication) -> None:
        self.__generator = Generator(exam_parameters)
        self.__com = communication
        self.__stagebutton = StageButton()

    def begin(self):
        self.__com.send(self.__generator.parameters.dumps())
        sleep(1)
        incoming = self.__com.read()
        logging.debug(f"Data received: {incoming}")
        if(self.checkAck(incoming)):
            logging.info("ACK received")
            return True
```

```
29          else:
30              return False
31
32      def checkAck(self, incoming):
33          myJson = json.loads(incoming)
34          key = 'ack'
35          if key in myJson:
36              return True
37          else:
38              return False
39
40      @property
41      def stagebutton(self):
42          return self.__stagebutton
43
44      def updateParameters(self, new_parameter: Parameters):
45          self.__generator.parameters = new_parameter
46          return self.begin()
47
48      def updateButtonStage(self, bs):
49          if(bs == "IDLE"):
50              self.__stagebutton.stage = Stage.IDLE
51          elif(bs == "PREPARE"):
52              self.__stagebutton.stage = Stage.PREPARE
53          elif(bs == "SHOOT"):
54              self.__stagebutton.stage = Stage.SHOOT
55          else:
56              logging.error(f'Wrong state: {bs}')
57              raise Exception
```

O construtor de Exam recebe um objeto de Parameters e um objeto de Communication, retorna None. É criado um objeto de Generator tendo como parâmetros o objeto de Parameters recebido pelo construtor e esse objeto de Generator é inserido no atributo *__gen* da classe Exam. O atributo *__com* é inicializado com o objeto comunication recebido por parâmetro. Para a inicialização de *__stageButton* é criado um objeto da classe StageButton.

O método *begin()* realiza o envio dos parâmetros para o *firmware* e fica à espera de uma resposta do tipo ACK (que representa uma resposta de recepção). Caso a mensagem recebida (que é do formato JSON) contenha a chave “ack” então é possível confirmar que o microcontrolador recebeu a mensagem, então *begin()* retorna ’True’. Para a conferência da presença do ’ACK’ na mensagem é utilizado o método

checkAck() que recebe a mensagem, transforma em JSON via *json.loads()*, verifica a presença da chave 'ack' e retorna 'True' caso encontre e 'False' caso não.

Assim como em outras classes, há um decorador de propriedade para servir como um *get* e, no caso do método stagebutton, é retornado o atributo *__stageButton* da classe. O método updateParameter() recebe por parâmetro um objeto da classe Parameters e atualiza *__gen.parameters* da classe Exam. Outro método de atualização é updateButtonStage(bs: *string*) que recebe uma *string* que representa o estado do botão e atribui o estado recebido ao *__stageButton* de Exam caso seja um estado válido, caso contrário levanta uma exceção de estado errado. Ter um método end() para enviar os parâmetros zerados para o *firmware* foi cogitado durante o desenvolvimento, porém é mais seguro o *firmware* ser encarregado de tal atividade, pois mesmo tendo precauções, ainda pode haver algum erro de comunicação. Então, sempre que um exame é encerrado (*flag done_tick* no *firmware*), os parâmetros são reinciados por lá. Mais detalhes sobre essa operação na seção de desenvolvimento do *firmware* 5.2.

5.1.2 Programa principal - *backend*

O programa principal pode ser dividido em duas categorias: sem interface e com interface. Geralmente o programa sem interface é somente um arquivo *"main.py"* que possui todas as classes e módulos necessários para realizar testes do funcionamento e sequência de chamadas. Com as saídas e comportamentos testados é possível migrar para uma interface gráfica. Portanto, antes do desenvolvimento da interface, o desenvolvimento do *backend* e do *firmware* devem trabalhar em paralelo para que haja coerência entre os programas. Nessa seção será discutido o funcionamento do programa *"main.py"* do *backend*.

```
1 #!/usr/bin/env python
2
3 import sys
4 import logging
5 from lib.communication import serialCom
6 from lib.generator import Parameters
7 from lib.exam import Exam
8 from lib.status import Status, State
9 from blessed import Terminal
10 from time import sleep
11
12 logging.basicConfig(format='%(asctime)s: %(levelname)s - %(message)
    ↪ s', level=logging.DEBUG, datefmt='%d-%b-%y %H:%M:%S')
13
14 def main():
```

```
15      status = Status()
16      status.status = State.STARTING
17      logging.info(f"Status: {status.description()}")
18      status.status = State.STAND_BY
19      logging.info(f"Status: {status.description()}")
20
21      portName = 'COM3'
22      baudRate = 115200
23      timeout = 1
24      kV = 149
25      mAs = 800.0
26      focus = "fine"
27      MAX_ATTEMPTS = 10
28
29      parameters = Parameters(kV, mAs, focus)
30      communication = serialCom(portName, baudRate, timeout,
            ↪ MAX_ATTEMPTS)
31      exam = Exam(parameters, communication)
32
33      if(communication.connect() == False):
34          if(communication.reconnect() == False):
35              sys.exit("Not possible to reconnect. Exiting program.")
36
37      sleep(1)
38      if(communication.handshake() == False):
39          logging.error("Bad handshake")
40          sys.exit("Bad handshake. Exiting")
41
42
43      status.status = State.WAITING
44      logging.info(f"Status: {status.description()}")
45
46
47      status.status = State.BUSY
48      logging.info(f"Status: {status.description()}")
49
50      if(exam.begin() == True):
51          logging.debug("Exam has started. ACK Received")
52
53          status.status = State.READY
54          logging.info(f"Status: {status.description()}")
```

```
55
56      else:
57          logging.debug("Ack not received")
58          sys.exit("Exiting")
59
60      while(1):
61          bs = communication.read()
62          if(bs == "Done"):
63              logging.info("Done")
64
65              break
66          else:
67              exam.updateButtonStage(bs)
68              logging.info(exam.stagebutton.stage.name)
69
70      communication.close()
71      sys.exit('Disconnected.')
72
73  if __name__ == "__main__":
74      main()
```

Após a importação de pacotes padrões do Python utilizados nessa aplicação, são importados também os módulos desenvolvidos comentados nas seções anteriores. Também é criada uma variável no escopo global para parametrizar a função de logging, responsável pelos *outputs* de *logs* de programa. Com esse pacote, é possível determinar qual tipo de informação deve ser vista em determinado momento, por exemplo *debug*, *info*, *warning* ou *error*. Também são descritas duas funções para auxiliar na visualização no terminal de comando: *drawTop()* e *reprint()*, mas essas são opcionais.

Na função *main()* é criado o objeto de Status da aplicação e demais variáveis de parâmetros para que sejam iniciados alguns objetos pertinentes ao exame. Ou seja, essas variáveis são entradas do programa que teriam influência direta do usuário e do computador (para objetos de conexão serial). Em sequência, são criados objetos de Parameters, serialCom e Exam. O objeto de Parameters é criado passando para o construtor valores de kV, mAs e foco. Para comunicação a construção é feita de maneira semalhante com o nome da porta, taxa de bits por transmissão, tempo máximo de resposta e a quantidade de tentativas possíveis para conectar e fazer o *handshake*.

Em seguida é feita a tentativa de conexão e, caso não seja possível, o método de reconexão é chamado e se, mesmo assim, não for possível então o programa é encerrado. Na interface gráfica esse comportamento será modelado para voltar para a página inicial e tentar novamente com outra porta. Logo, tenta-se realizar uma troca de

mensagens *handshake*, previamente explicada. Caso o *handshake* não tenha sucesso depois de MAX_ATTEMPS tentativas, então o programa é encerrado. Novamente isso será tratado para voltar à página inicial na interface gráfica.

Nesse momento o exame é iniciado através do método *exam.begin()* que deve retornar True caso tenha recebido um ACK do *firmware*, caso contrário o programa também é encerrado. Em todas essas abordagens em que o programa é encerrado, é necessário fazê-lo pois é importante garantir que o objeto intermediário entre o *software* e o gerador, nesse caso o *firmware*, tenha se conectado com sucesso.

Além disso, é importante que consiga fazer uma troca de mensagens completa, isenta de erros, que envie os parâmetros completos e que tenha certeza que esse parâmetro chegou. Caso uma dessas condições não seja satisfeita, a conexão é encerrada, junto com o programa. Assim é possível realizar uma nova tentativa.

Quando os parâmetros de exame são enviados e recebidos pelo *firmware* (com garantia) então o programa segue normalmente, ou seja, é feito um laço *while* para receber o estado do botão. Há uma condição nesse laço que é: caso a mensagem vinda do *firmware* seja a mensagem “Done”, então o exame foi concluído e o programa pode ser encerrado; caso contrário, o programa continua lendo o estado do botão e informando o usuário do mesmo. Ao sair do laço *while*, por fim, é realizada uma desconexão da porta serial e encerrado o programa.

Esse ciclo de programa da criação do objeto Exam até o “Done” é o ciclo para um exame. Na interface o usuário terá três possibilidades de acordo com o Diagrama de Estados visto na Seção 4.4: Repetir o exame com os mesmos parâmetros, realizar um novo exame com novos parâmetros ou sair do programa. O que muda do programa de *backend* para o *frontend*, basicamente, é a ordem das chamas e criação de objetos. Portanto, o desenvolvimento desse programa *backend main* é o cerne do exame, que pode ser adequado para um dinamismo melhor com o usuário.

5.2 *FIRMWARE*

O *firmware* é o desenvolvimento do código embarcado no microcontrolador. Ele é desenvolvido em C++ com algumas bibliotecas e o framework do Arduino. Essa seção descreve as classes, funções e arquivos auxiliares e também o programa principal de execução, com *setup* e *loop*.

5.2.1 Classes

Assim como no desenvolvimento do *software*, no *firmware* também é utilizada programação orientada a objetos. Nesse sentido, foram criadas cinco classes para representar objetos do programa. As classes são Message, Control, Button, Generator e StageButton.

Ademais, há alguns arquivos auxiliares para auxiliar nas definições e outras funções para tratar de timers utilizados. Todas as classes e arquivos serão descritos nessa seção. Os arquivos do tipo *header* (.h) foram alocados na pasta “include” do *firmware* e todas as implementações estão na pasta src. Isso é feito para melhor organização do programa.

5.2.1.1 Button

A classe Button é uma classe para representar um botão genérico que possui um pino, um estado, uma *flag* para saber se foi pressionado, uma variável de controle para fazer o *debounce* do botão e um id.

O construtor atribui os pinos e id ao botão e inicializa as demais variáveis em valor falso, que é o padrão. Além disso, também configura o pino recebido como entrada com resistor de *pullup* interno do microcontrolador.

O método wasPressed() apenas retorna a *flag* correspondente. O método reset() torna a *flag* _wasPressed como falso e reinicia _startDebounce. Também há métodos do tipo *getter* e *setter* para os atributos de pino e id do botão.

Ademais, a classe possui um método poll() que recebe o valor de tempo no instante atual e atualiza o estado do botão, tendo como referência a leitura feita do pino correspondente. Além disso, o método faz um debouce do botão para garantir que não haja algum erro de interpretação ao pressionar o botão devido à variação mecânica do contato do botão físico.

```
1  #ifndef BUTTON_H
2  #define BUTTON_H
3
4  #include <Arduino.h>
5  #include "pinout.h"
6
7  class Button {
8      protected:
9          uint8_t _pin;
10         boolean _state;
11         boolean _wasPressed;
12         uint16_t _startDebounce;
13         String _id;
14
15     public:
16         Button(uint8_t pin, String id);
17         uint16_t debounceMs;
18         boolean wasPressed(void);
19         void reset(void);
```

```
20          void poll(uint16_t now);
21          void buttonAction(uint8_t outputPin);
22          void setPin(uint8_t pin);
23          void setId(String id);
24          String getId(void) const;
25          uint8_t getPin(void) const;
26 };
27 #endif
```

5.2.1.2 *StageButton*

A classe StageButton é uma classe contâiner que possui dois botões como atributos, além de variáveis de controle como *_stage*, *_enable* e *_ready2shoot*.

A classe possui como um membro público uma enumeração chamada Stage para representar os estados do botão, assim como discutido no *software*. Também possui métodos auxiliares.

O construtor inicia o estado do botão como IDLE e desabilitado. Também são iniciados os construtores da classe Buttons para seus dois objetos de botão. Além disso, todos os pinos de saída atrelados ao botão são iniciados em nível lógico baixo.

Há na classe métodos *get* e *set* para _stage e métodos para habilitar, deseabilitar e verificar se o botão está habilitado - *enable()*, *disable*, *isEnable()* respectivamente.

O método *writeStage()* verifica o valor do estado *_stage* e faz o controle de liga e desliga dos pinos associados à esse estado. Por exemplo, caso o estágio do botão seja PREPARE, então o pino associado ao PREPARE é acionado em nível lógico alto e os demais pinos em nível lógico baixo. A mesma lógica é aplicada para os demais estágios do botão.

Por fim, tem-se um método para tornar o estágio do botão baixo, que atribui IDLE para *_stage* e escreve nível lógico baixo em todos os pinos. Esse método é utilizado para garantir que ao final do exame, não haverá nenhum sinal lógico alto nos pinos de estágio.

```
1 #ifndef STAGE_BUTTON_H
2 #define STAGE_BUTTON_H
3
4 #include <Arduino.h>
5 #include "button.h"
6 #include "pinout.h"
7
8 namespace stb{
9     class StageButton{
```

```
10          public:
11              enum class Stage{
12                  IDLE,
13                  PREPARE,
14                  SHOOT
15              };
16              StageButton(Button *button1, Button *button2);
17              Stage getStage(void) const;
18              void update(uint16_t now);
19              void setStage(Stage new_stage);
20              bool isEnable(void);
21              void enable(void);
22              void disable(void);
23              void writeStage(void);
24              void toIdle();
25          private:
26              Stage _stage;
27              bool _enable;
28              bool _ready2shoot;
29              Button *b1;
30              Button *b2;
31      };
32  }
33  #endif
```

5.2.1.3 *Message*

A classe Message possui três atributos privados: um documento estático JSON e dois métodos privados (*error_handeling* e *stringfy*). Como atributos públicos dessa classe têm-se os métodos *receive*, *handshake*, *ack*, *updateStatus* e *getMessage*.

O método *receive* tem como objetivo a leitura da porta serial e destinação correta em pacotes JSON. Inicia-se o método fazendo a leitura serial e verificando se a mensagem é uma requisição de ACK através da mensagem “ID request”. Caso seja uma requisição, é chamado o método *handshake* e retornado false para quem chamou.

Caso não seja um pedido de *handshake*, a mensagem é deserializada, atribuída ao documento JSON privado da classe e retornado true para o chamador do método. O método *handshake*() é curto, ele apenas responde com a palavra passe de comunicação “Handshake-ID:CIXP”

Já o método *ack()* cria um novo dicionário composto pelo documento interno da classe (m_doc) adicionado da chave “ack” com valor 1 para representar que recebeu com sucesso as mensagens. Em seguida, é serializada essa mensagem e enviado por

serial para o *software*, que a espera para dar continuidade ao programa.

O método updateStatus() é responsável por enviar para o *software* a informação do estado atual do botão. Para isso é utilizado o método *stringfy()* para transformar o objeto de StageButton em uma *string* válida de acordo com o estado do botão (IDLE, PREPARE ou SHOOT). Caso tenha algum problema de comunicação e não seja um desses três estados, é enviado uma *string* ERROR STAGE.

Por fim, o método error_handeling envia para o *software* um dicionário com o motivo e o código de erro caso não tenha conseguido deserializar a mensagem inicial recebida por *receive()*. Além desse método, há também um último método que é o getMessage() que apenas retorna o documento JSON.

```
#ifndef MESSAGE_H
#define MESSAGE_H

#include <Arduino.h>
#include <ArduinoJson.h>
#include "stagebutton.h"

const uint8_t doc_len = 128;

class Message{
    public:
        bool receive();
        void handshake();
        void ack();
        void updateStatus(stb::StageButton st);
        StaticJsonDocument<doc_len> getMessage()const;

    private:
        StaticJsonDocument<doc_len> m_doc;
        void error_handeling(const char *error);
        String stringfy(stb::StageButton st);
};

#endif
```

5.2.1.4 Generator

A classe Generator representa o gerador, porém do lado do *firmware*. Tem como atributos privados kV, mAs e focus. Também possui método públicos *getkV()*, *getmAs()* e *getFocus()* além do construtor.

O construtor recebe o documento JSON que contém os parâmetros, separa e atribui os valores às variáveis utilizando a estrutura JSON. Na parte de atribuição de foco é verificado se a palavra recebida é "*thick*" representando foco grosso, e caso seja, é atribuído valor booleano falso para o atributo focus. Caso contrário, é atribuído "true".

Os métodos restantes são apenas *getters* que retornam os valores privados de atributos kV, mAs, e focus. Ademais, a classe necessita incluir o arquivo "message.h". Apesar de ser uma classe mais curta, é importante para representar o gerador.

```
#ifndef GENERATOR_H
#define GENERATOR_H

#include <Arduino.h>
#include <ArduinoJson.h>
#include "message.h"

class Generator{
    public:
        Generator(StaticJsonDocument<doc_len> doc);
        uint8_t getkV();
        float_t getmAs();
        bool getFocus();
    private:
        uint8_t kV;
        float_t mAs;
        bool focus;
};

#endif
```

5.2.1.5 Control

A classe Control é a classe mais importante do *firmware* porque é ela que converte os valores de kV para um valor que será enviado para a função que escreve a saída PWM no pino associado.

```
#ifndef CONTROL_H
#define CONTROL_H

#include "Arduino.h"
#include "generator.h"
#include "definitions.h"

class Control{
```

```
9     public:
10        Control();
11        void getPWM_values(Generator g);
12        void writeParameters();
13        void resetParameters();
14    private:
15        void write_kV(uint16_t pwm_value);
16        void write_mAs(uint16_t pwm_value);
17        void write_focus();
18        uint16_t convertkV_PWM(uint8_t input, uint8_t lo_var,
              ↪ uint8_t hi_var,
19             float_t lo_ref, float_t hi_ref, uint8_t resolution);
20        uint16_t convertmAs_PWM(float_t input, float_t lo_var,
              ↪ float_t hi_var,
21             float_t lo_ref, float_t hi_ref, uint8_t resolution);
22        uint16_t kV_PWM;
23        uint16_t mAs_PWM;
24        bool focus_output;
25 };
26 #endif
```

5.2.1.5.1 Canais PWM

PWM (do inglês *Pulse Width Modulation*) é a modulação por largura de pulso e a tensão de saída é proporcional à média de tensão entre o tempo ligado e desligado do período. Com PWM, pode-se variar a tensão de saída conforme o parâmetros de entrada.

O PWM depende da frequência de trabalho, resolução de bits e porcentagem de ciclo de trabalho (do inglês duty cycle). O microcontrolador ESP32 tem possibilidade de trabalhar com valores de clock de 80MHz à 240Mhz. entretanto o recurso “led pwm” tem frequência máxima de 80Mhz.

Outro parâmetro de PWM é a resolução. Ela também é um fator limitante da frequência que consegue-se atingir na prática. É possível encontrar a frequência máxima através da equação:

$$F_m = \frac{Clock}{Resolucao} \tag{5}$$

A resolução escolhida é 10bits, então deve-se trabalhar com valores PWM até 2^{10} valores, ou seja, de 0 a 1024. Para essa resolução, a frequência máxima calculada por meio da equação 5 é de 80MHz/1024 = 80MHz/1024 = 78,125kHz. Por esse motivo,

a frequência escolhida foi de 30kHz. A tensão de saída desejada é dada através da equação 6 abaixo:

$$V_{out} = Vcc \times duty\_cycle \tag{6}$$

Onde Vcc é a tensão medida no pino de saída PWM ao enviarmos 100% do *duty_cycle* (ou seja 1024) na função interna ledcWrite do ESP32. Mais detalhes sobre essa calibração na Seção 5.2.2. Então, tendo o valor de tensão de saída desejado, é possível determinar o *duty_cycle* correspondente, assim multiplicar esse valor de *duty_cycle* (que é em porcentagem) pela resolução (1024 para 10bits). Assim, o valor resultante será o valor enviado como parâmetro para a função *ledcWrite()*. Essa função, por sua vez, realiza o ciclo PWM no canal atribuído e esse canal possui um pino atrelado que irá aprensentar a tensão média correspondente.

5.2.1.5.2 Métodos públicos da Classe Control

O construtor de Control inicia os valores *kV_PWM*, *mAs_PWM* em zero e *focus_output* em nível baixo. Também atrela o pino de saída que representa o kV (KV_PIN) ao canal 0 do PWM e atribui ao canal 0 frequência FREQ_HZ e resolução RESOLUTION. Todas essas constantes são definidas em “pinout.h” e “definitions.h”.

O mesmo acontece para o pino de mAs (MAS_PIN), porém esse pino é atribuído ao canal 2 e esse é configurado com a mesma frequência e resolução anterior.

Por último, no construtor é configurado o pino FOC_PIN como modo OUTPUT. Note que todos os pinos e constantes podem ser alterados nos arquivos “pinout.h” e “definitions.h” dentro da pasta “include” e no Apêndice D.

O método *getPWM* utiliza os métodos *convertkV_PWM() convertmAs_PWM()* e *getFocus()* para atribuir os valores aos atributos privados correspondentes da classe.

O método público *writeParameters()* apenas usa os métodos privados *write_mAs*, *write_kV* e *write_focus* e coloca a variável *focus_output* de acordo com o recebido. Já o método *resetParameters()* faz algo semelhante, porém coloca todos os valores para os estados padrões (zero e false).

5.2.1.5.3 Métodos privados da Classe Control

Os métodos *writekV* e *writemAs* são responsáveis por receber o valor PWM e escrever nos respectivos canais. Algo semelhante acontece com *write_focus*, que é escrito nível lógico alto ou baixo no pino FOC_PIN dependendo do valor do atributo *focus_output*.

Os métodos *convertkV_PWM()* e *convertmAs_PWM()* são responsáveis por converter o valor da variável original (kV ou mAs) em valores que serão inseridos no método *writekV()* e ,por consequência, em *ledcWrite()*. O que difere de um método ou outro é apenas o tipo de variável que será recebida como parâmetro. Enquanto *convertkV_PWM()* recebe inteiros como entrada, *convertmAs_PWM()* recebe valores de ponto flutuante na entrada, entretanto a operação em ambos métodos é igual. Dentro do método de conversão, o objeto a ser convertido (kV ou mAs) se chamará apenas 'var'.

Com a finalidade de linearizar o comportamento de saída, tendo como base a tabela no Anexo A, são necessários os valores máximo e mínimos de entrada e os valores máximos e mínimos das saídas (tensão de referência). Dessa forma, caso os valores de correspondência entre entrada e saída mudem de acordo com o gerador, é possível apenas ajustar os valores em "definitions.h" e o programa continua a funcionar. Para isso, é importante que o comportamento seja considerado linear, caso contrário é necessário remodelar a equação dentro do método de conversão.

Outro detalhe é que para a variável mAs não há disponível na documentação do gerador elétrico quais são as tensões de saída correspondentes para cada valor de mAs. Então os valores máximos e mínimos da tensão de referência são os mesmos de kV (0,77 e 2,77). Entretanto eles podem ser ajustados de acordo com a necessidade, como comentado anteriormente.

O método de conversão inicia calculando a variação dos máximos e mínimos de entrada e saída. Logo após, são feitas algumas verificações e, em seguida, segue-se para o cálculo. O cálculo é dividido em três partes e nessa explicação será usado de exemplo o parâmetro de kV, porém o mesmo cálculo acontece para mAs.

A primeira é a conversão de kV para V_{ref} e segue um comportamento linear descrito pela Equação 7 abaixo, assim os valores de kV são mapeados para V_{ref} de saída:

$$out = (in - kV_{min}) \cdot 0.01818 + V_{ref_{min}} \tag{7}$$

A segunda equação é a conversão do valor V_{ref} calculado acima para a porcentagem de *duty_cycle* que aquele valor representa em uma escala de 0 a VCC (calibrado). Essa cálculo é feito na equação 6 descrita anteriormente.

A última conversão é mais simples, é apenas multiplicado esse valor de *duty_cycle* (que é em porcentagem) pela resolução 1024 ou 2^{10}. Então, tem-se o valor final que é passado para *ledcWrite()*. Para facilitar a implementação, esses três passos de conversão são sintetizados em uma única equação e essa é utilizada na implementação para uma conversão direta entre entrada (kV ou mAs) para valor que é passado para a função interna de PWM. Essa equação está descrita na Equação 8.

$$output = \left[\frac{(input - var_{min}) \cdot \Delta V_{ref} + V_{ref_{min}} \cdot \Delta var}{V_{ref_{max}} \cdot \Delta var} \right] \cdot 2^{resolution} \quad (8)$$

5.2.2 Arquivos auxiliares

Além dos arquivos de cabeçalhos de classe (.h) e implementações (.cpp), também há dois arquivos para o auxílio das operações, são eles: “definitions.h” e “pinout.h”. Como o próprio título descreve, o arquivo *definitions* contém a definição de algumas constantes enquanto o *pinout* se refere à definição dos pinos utilizados. Ambos arquivos são mostrados no Apêndice D.

Em “definitions.h” há informações como frequência de PWM, canais PWM utilizados, máximos e mínimos de valores para kV, mAs e tensão de referência de saída para ambos. Essas constantes podem ser ajustadas conforme comportamento do gerador para tais parâmetros.

Além disso, há uma parte que define o valor de VCC utilizado nas contas nos métodos *convertmAs_PWM()* e *convertkV_PWM()*. Note que para o VCC utilizado no cálculo de kV há dois valores, com e sem LED acoplado, e o mesmo ocorre para o VCC de mAs. Isso porque há diferença de tensão ao ter um LED conectado à saída PWM, então para que o valor de saída seja correto, é necessário que seja considerada a queda de tensão com o LED.

Outro detalhe é que, mesmo sem o LED para ambos casos (kV e mAs), o valor de VCC não é 3,3V, mas um pouco menos - em torno de 3,2V. Esse valor deve ser calibrado ao conectar o *hardware* pela primeira vez. Essa calibração é feita enviando no canal PWM 100% do *duty_cycle*, ou seja, enviando 1024 em *ledcWrite*. Dessa forma, é possível medir com um multímetro qual a tensão máxima que aquele canal naquele pino consegue atingir. Esse valor deve, então, ser atribuído às constantes correspondentes nesse documento.

Caso um LED seja conectado aos pinos de saída PWM, além da calibração desse valor com 100% do *duty_cycle* no PWM, também é necessário mudar a constante utilizada no método de conversão. Por exemplo, caso seja conectado um LED ao pino KV_PIN então calibra-se a constante VCC_KV_WITH_LED e essa constante deve ser utilizada no método convertkV_PWM.

5.2.3 Programa principal

O programa principal (main.cpp) é separado em quatro partes: Escopo global, *setup*, *loop* e demais funções. Essas quatro partes são explicadas nessa subseção.

5.2.3.1 *Escopo global*

No escopo global, são criados objetos de algumas classes que são utilizados no decorrer do programa, esses objetos precisam estar acessíveis a todas as funções desse arquivo.

Além dos objetos, também são declaradas algumas variáveis (*flags* em geral) que auxiliam na sincronização de alguns processos. Nessa parte, ainda, são declarados cabeçalhos das demais funções desse programa.

```
#include <Arduino.h>
#include <ArduinoJson.h>
#include "message.h"
#include "pinout.h"
#include "button.h"
#include "stagebutton.h"
#include "control.h"

using namespace stb;

Button button1(STAGE1, "Stage 1");
Button button2(STAGE2, "Stage 2");
StageButton stagebutton(&button1, &button2);
Message mes;
Control control;

bool done_tick = false;
bool first_shoot = false;
bool first_prepare = false;
bool prepare_done = false;
hw_timer_t * timer_shoot = NULL;
hw_timer_t * timer_prepare = NULL;

typedef StageButton::Stage st;

void IRAM_ATTR onTimerShoot();
void IRAM_ATTR onTimerPrepare();
void startTimerShoot(uint64_t endtime_ms);
void startTimerPrepare(uint64_t endtime_ms);
void stopTimerShoot();
void stopTimerPrepare();
```

5.2.3.2 *Setup*

A função de *setup* é executada apenas uma vez quando o processo é iniciado e nela é iniciada a comunicação serial com 115200 de *baudrate*, atribuído o valor de *debounce* dos dois botões e por fim, alterado o modo dos pinos de saída da botão para OUTPUT.

```
void setup(){
    Serial.begin(115200);
    button1.debounceMs = 200;
    button2.debounceMs = 200;
    pinMode(IDLE_OUTPUT, OUTPUT);
    pinMode(PREPARE_OUTPUT, OUTPUT);
    pinMode(SHOOT_OUTPUT, OUTPUT);
}
```

5.2.3.3 *Loop*

A função *loop* tem como característica a execução cíclica e nela todo processo do exame é feito. O código abaixo refere-se à função *loop()* implementada. Em sequência há uma explicação para cada parte dessa função.

```
void loop(){
    if(Serial.available()){
        if(mes.receive() == true){
            mes.ack();
            Generator gen(mes.getMessage());
            control.getPWM_values(gen);
            control.writeParameters();
            delay(500);
            stagebutton.enable();
        }
    }
    uint16_t now = millis();
    stagebutton.update(now);

    if(stagebutton.isEnable()){
        if(stagebutton.getStage() == st::PREPARE){
            if(first_prepare == false){
                startTimerPrepare(PREPARE_MIN_MS);
                first_prepare = true;
            }
            mes.updateStatus(stagebutton);
```

```
22              stagebutton.writeStage();
23          }
24
25          if(stagebutton.getStage() == st::SHOOT){
26              if(prepare_done == true){
27                  if(first_shoot == false){
28                      startTimerShoot(SHOOT_MAX_MS);
29                      first_shoot = true;
30                  }
31                  mes.updateStatus(stagebutton);
32                  stagebutton.writeStage();
33              }else{
34                  stagebutton.setStage(st::PREPARE);
35                  mes.updateStatus(stagebutton);
36                  stagebutton.writeStage();
37              }
38          }
39
40          if(stagebutton.getStage() == st::IDLE){
41              mes.updateStatus(stagebutton);
42              stagebutton.writeStage();
43              if(first_prepare == true){
44                  stopTimerPrepare();
45              }
46              first_prepare = false;
47              first_shoot = false;
48          }
49          delay(50);
50      }
51
52      if(done_tick == true){
53          control.resetParameters();
54          Serial.println("Done");
55          done_tick = false;
56      }
57  }
```

A execução começa verificando se há alguma mensagem no buffer serial, caso contenha, o método *receive()* é chamado e caso tenha retorno seja verdadeiro, então o método *ack()* é chamado para enviar a resposta de recebimento ao *software*. Lembrando, nesse momento de chamada de *receive()* também é chamado o método *handshake*() internamente.

Após enviar o ACK ao *software*, é criado um objeto do tipo gerador com a mensagem recebida e, logo após, são extraídos os valores de PWM utilizando o método de Control “getPWMvalues”. Com os valores extraídos para os atributos privados de Control, são escritos esses valores nas saídas PWM utilizando *control.writeParameters()*. Então é feito um atraso de 500ms e o botão de dois estágios é liberado.

Nesse momento, é atualizado o estado do botão por meio do método *update()* de StageButton e, em seguida, há uma guarda de ação, onde é verificado se o botão já foi liberado. Caso positivo, é verificado qual o estado do botão e separada a execução de acordo com esse estado.

Caso o estado seja PREPARE, é verificado se é a primeira vez que a execução passa por ali, e ,em caso positivo, é iniciado um *timer* de PREPARE_MIN_MS e colocada a *flag* de primeira vez em *true*. Caso não entre nessa execução de primeira vez de PREPARE, então é apenas enviado o estágio do botão para o *software* e é atualizado o pino de saída para desse estágio.

Como um *timer* foi iniciado na primeira vez em que ocorre um evento de PREPARE, assim que o *timer* chega ao fim, é executada a função *onTimerPrepare()* que verifica se o último estágio registrado ainda foi PREPARE e, em caso afirmativo, é atribuído à *flag prepare_done* valor verdadeiro, para a *flag first_prepare* o valor falso e o *timer* é excluído com o método *stopTimerPrepare()*.

Já se na primeira verificação o último estágio registrado não for de PREPARE, então ambas *flags* recebem valor falso e o *timer* também é parado. Esse *timer* tem objetivo de garantir que o usuário se mantenha por um tempo mínimo no estágio PREPARE antes de realizar o disparo (SHOOT). É necessário garantir esse tempo mínimo, definido por PREPARE_MIN_MS, pois há um tempo mínimo para o ânodo giratório atingir a velocidade adequada e, também, para que o filamento seja pré-aquecido, dado que há uma inércia térmica relacionada ao aquecimento do filamento.

Na segunda guarda do escopo de “if(stagebutton.isEnable())” é verificado se o estágio pressionado é SHOOT e em caso afirmativo, é verificado se o usuário pressionou por um tempo mínimo o estágio PREPARE - através da *flag* prepare_done. Em caso positivo, é verificado se é a primeira vez que a execução entra nessa guarda e então é iniciado o *timer* de disparo caso seja a primeira vez. Caso não seja a primeira vez, o programa simplesmente envia para o *software* o estágio SHOOT do botão e atualiza o pino de saída do estágio SHOOT.

Caso o tempo de preparo ainda não tenha sido satisfeito, mas o botão pressionado é de SHOOT, então o estágio se mantém no estágio PREPARE, para garantir o funcionamento correto. Nessa parte também é atualizado para o *software* e para o pino de saída o estágio PREPARE.

Caso o *timer* de disparo chegue ao tempo definido por SHOOT_MAX_MS, é

chamada a função onTimer() que é responsável por desativar o botão de dois estágios, escrever as saídas do pino para nível baixo, colocar a *flag done_tick* para *true* e encerrar o *timer*. Esse momento significa que o usuário já disparou por tempo suficiente o raio-X e o exame é encerrado.

Por fim, a última guarda dentro de "if(stagebutton.isEnable())" verifica se o estágio pressionado é IDLE e então é atualizado o estágio para o *software* e atualizado o pino de saída IDLE para nível alto. Para não continuar incrementando o *timer* de *prepare* quando o usuário esteve em PREPARE ao menos uma vez e volta para IDLE, então nesse momento é parado o *timer* de *prepare*. Além disso, as *flags* de primeira entrada em PREPARE e SHOOT são reiniciadas para falso pois é necessário que ocorram os tempos mínimos de PREPARE e SHOOT novamente.

Saindo da condição de botão pressionado, ou seja, voltando para a execução cíclica, é verificado se o exame foi concluído por meio da *flag* "done_tick". Caso o exame tenha chegado ao fim, então são reiniciados os parâmetros - colocados todos os valores para o estado inicial (zero e/ou baixo). Também é enviado para o *software* uma mensagem de "Done" e então a execução é encerrada e o exame foi concluído. O *software* é encarregado por encerrar a comunicação serial.

5.2.3.4 Demais funções

Além das funções de *loop* e *setup*, outras funções auxiliam na execução do programa e todas essas funções são relacionadas aos *timer* de disparo e *timer* de preparo.

As funções onTimerShoot() e onTimerPrepare() são chamadas quando os timers correspondentes estouram. Já as funções startTimerShoot() e startTimerPrepare() são responsáveis por iniciar os timers, atrelar a função chamada quando o mesmo chega ao fim, configurar o tempo de disparo e habilitar o *timer* correspondente. As funções de stopTimerShoot() e stopTimerPrepare() encerram os timers e apontam os ponteiros do *timer* para *null*, ou seja, desativa e exclui o *timer*.

```
1  void IRAM_ATTR onTimerShoot(){
2      stagebutton.disable();
3      stagebutton.toIdle();
4      done_tick = true;
5      stopTimerShoot();
6  }
7
8  void IRAM_ATTR onTimerPrepare(){
9      if(stagebutton.getStage() == st::PREPARE){
10         prepare_done = true;
11         first_prepare = false;
```

```
12          stopTimerPrepare();
13      }else{
14          first_prepare = false;
15          prepare_done = false;
16          stopTimerPrepare();
17      }
18  }
19
20  void startTimerPrepare(uint64_t endtime_ms){
21      timer_prepare = timerBegin(1, 80, true);
22      timerAttachInterrupt(timer_prepare, &onTimerPrepare, true);
23      timerAlarmWrite(timer_prepare, endtime_ms*1000, false); // ms
            ↪ to us
24      timerAlarmEnable(timer_prepare);
25  }
26
27  void startTimerShoot(uint64_t endtime_ms){
28      timer_shoot = timerBegin(0, 80, true);
29      timerAttachInterrupt(timer_shoot, &onTimerShoot, true);
30      timerAlarmWrite(timer_shoot, endtime_ms*1000, false); // ms to
            ↪ us
31      timerAlarmEnable(timer_shoot);
32  }
33
34  void stopTimerShoot(){
35      timerEnd(timer_shoot);
36      timer_shoot = NULL;
37  }
38
39  void stopTimerPrepare(){
40      timerEnd(timer_prepare);
41      timer_prepare = NULL;
42  }
```

5.3 *FRONTEND* DE *SOFTWARE*

A interface gráfica de usuário é a responsável por interagir com o usuário e repassar para o programa as informações necessárias para o funcionamento correto da aplicação. Nesse âmbito de desenvolvimento, a interface gráfica é chamada de *frontend* do *software*.

O *frontend* é desenvolvido em Python utilizando a biblioteca PyQt5 em sua

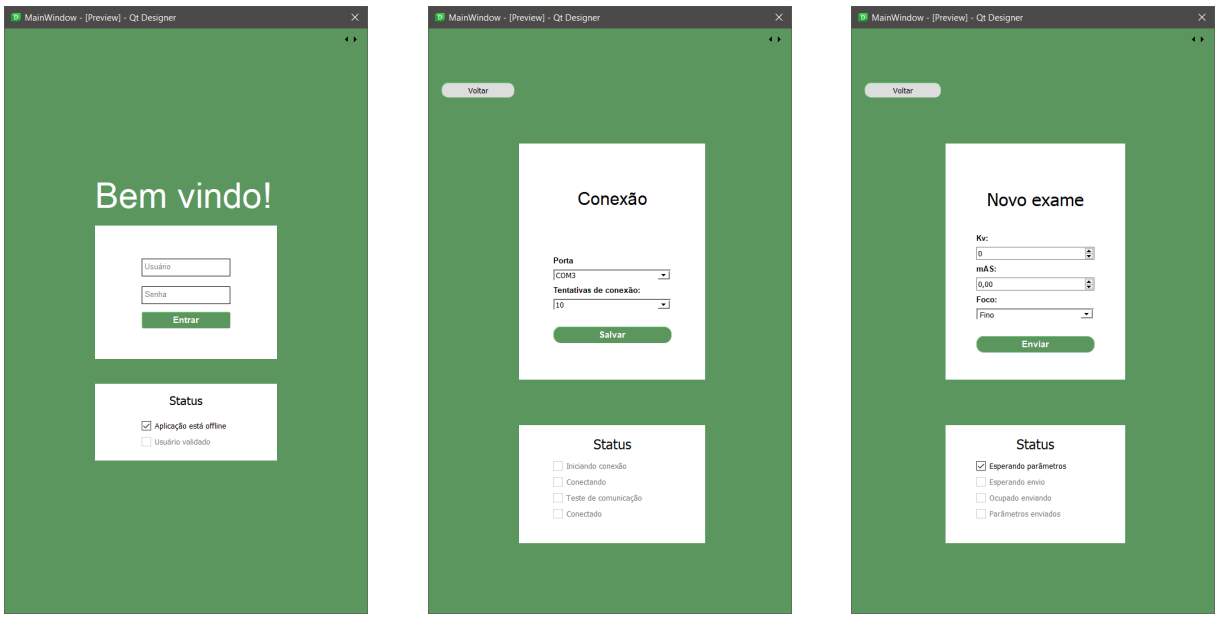
versão 5.15.4 com auxílio do ambiente de *design* (pyqt5-tools 5.15.4.3.2). Inicialmente as telas são desenhadas no Designer e a classe que é gerada automaticamente é herdada para a construção da aplicação final. Assim todos os métodos da interface podem ser utilizados com herança de classe.

Após a criação da classe de interface herdada da classe gerada pelo PyQt Designer, são criados os métodos de interface. Nesses métodos de eventos, são criados e manipulados os objetos do *backend* do *software* para que o exame seja realizado e as interações com o usuário aconteçam em uma ordem específica bem definida.

5.3.1 Layout e funcionamento

Para a realização do exame foram criadas cinco páginas com as quais o usuário irá interagir. As páginas foram criadas sob o recurso “stacked widget” do PyQt. Isso permite a navegabilidade entre páginas definida por código, ou seja, sempre que o usuário clicar em um determinado botão, o código irá lidar com aquele evento e chamar as janelas certas no momento certo, realizando operações em plano de fundo. As cinco telas são demonstradas nas Figuras 17 e 18.

Figura 17 – Telas da interface gráfica de usuário (1).



(a) Tela 1 - Acesso.

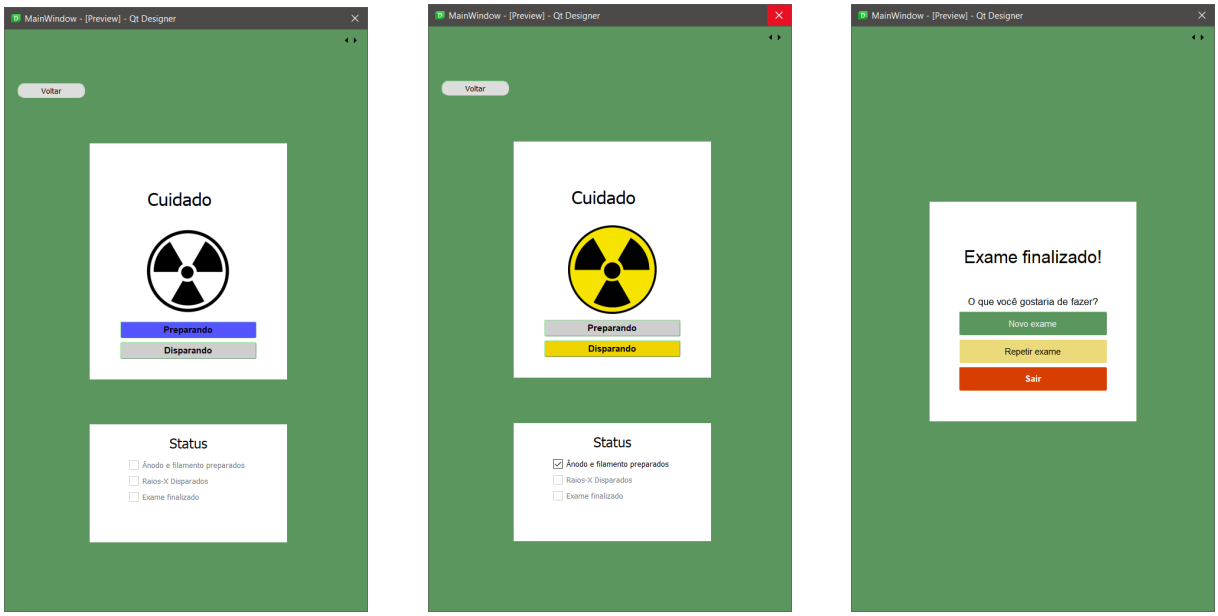
(b) Tela 2 - Conexão.

(c) Tela 3 - Envio de parâmetros.

Fonte: Elaboração própria.

A primeira página é apenas uma página de autenticação simples que, por ora, não foi conectada à nenhum banco de dados para validação de usuário, apenas é comparado um usuário e senha padrão com o inserido do usuário.

Figura 18 – Telas da interface gráfica de usuário (2).



(a) Tela 4 - Preparo de exame.

(b) Tela 4 - Disparo de raios-X.

(c) Tela 5 - Fim de exame.

Fonte: Elaboração própria.

Com o *login* validado, o usuário é levado à tela dois onde ele deve selecionar a porta COM onde o dispositivo está conectado e então o usuário clica em conectar e nesse momento os métodos responsáveis pela conexão são chamados.

Depois da aplicação conectada ao dispositivo, na terceira página, o usuário deve inserir os parâmetros para a realização do exame. Com todos os parâmetros inseridos corretamente, o usuário poderá clicar em “Enviar” para dar início à criação dos objetos Parameters e Exam, assim como visto na Subseção 5.1.1.5. Nessa tela ainda, é mostrado o estado em que a aplicação se encontra. Caso o usuário selecione voltar, então o exame e cancelado e a aplicação volta para a tela de conexão (tela dois).

Após o envio, a tela seguinte mostra o estado do botão de dois estágios. Sempre que o estágio do botão físico mudar, essa informação é recebida pelo *backend* e passada ao *frontend* para que o mesmo atualize as etiquetas de estágio corretas. Nessa tela também é mostrado o estado da aplicação. Caso o usuário escolha voltar, então o exame é reiniciado, os parâmetros são reiniciados e o usuário retorna para a tela de parâmetros (tela três).

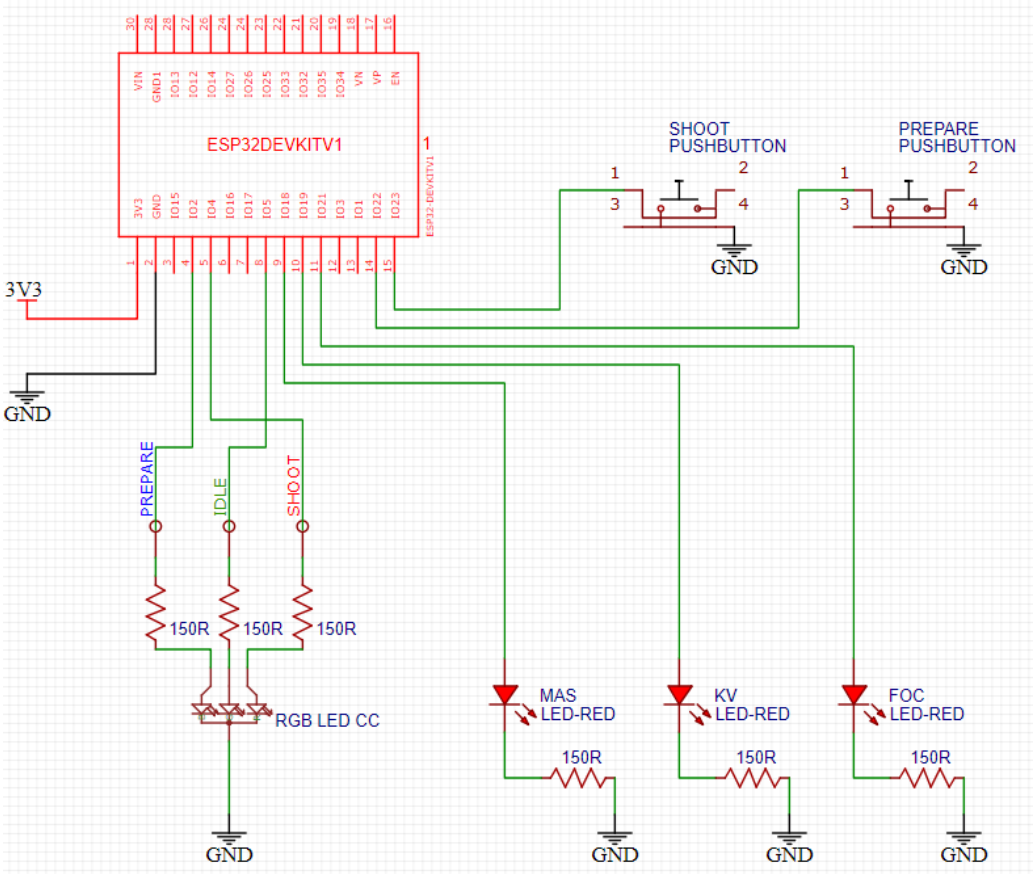
Por fim, ao realizar o exame o usuário é levado para a tela cinco onde é possível escolher entre fazer um novo exame, repetir um exame ou sair. Ao escolher novo exame, os objetos relacionados ao exame são destruídos e novos objetos com novos parâmetros são criados. Outra situação que o usuário pode selecionar é repetir o exame. Nesse caso o objeto de exame é destruído e criado um novo com os parâmetros anteriores. O último cenário é onde o usuário decide sair da aplicação, nesse caso o

exame é cancelado e o objeto de Exam é destruído e o usuário volta para a tela inicial, repetindo todo o ciclo.

5.4 HARDWARE

Essa seção mostra apenas a montagem do *hardware* para o protótipo em desenvolvimento e o esquemático pode ser contemplado na Figura 19 abaixo. Os itens necessários para a montagem são mostrados no capítulo de Materiais e Métodos 4 . A placa de desenvolvimento com ESP32 é fixada à *protoboard* e um cabo USB micro é conectado a essa placa e ao computador.

Figura 19 – Esquemático da montagem do *hardware*



Fonte: Elaboração própria.

A fonte de alimentação também é fixada à *protoboard*. É feita a conexão da fonte com o microcontrolador através das portas 3V3 e GND do microcontrolador. Todos os LEDs conectados ao microcontrolador têm seu outro terminal “GND” conectado em série com um resistor de 150Ω e ao trilho negativo da *protoboard*.

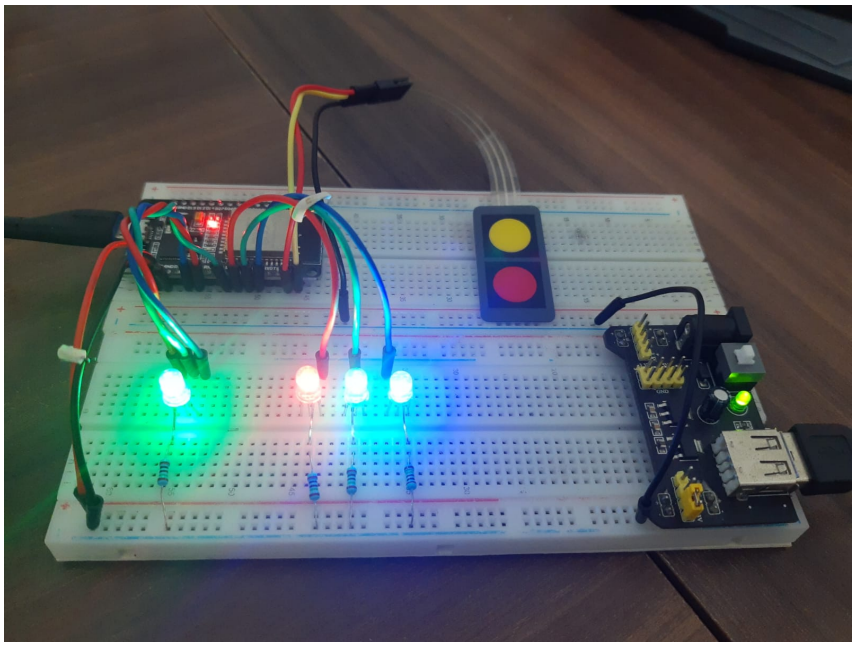
O LED que representa a saída do botão de dois estágios é conectado aos pinos correspondentes descritos em “pinout.h” do *firmware* assim como os pinos de saída de kV e foco.

A última conexão é do botão de dois estágios que é conectado ao microcontrolador pelos pinos descritos em “pinout.h”. O terminal negativo desse botão é conectado também ao trilho negativo da *protoboard*.

Como os LEDs são de três cores (vermelho, verde e azul), cada uma das saídas é atrelada a uma das cores. No LED que representa a saída dos estados, o estado IDLE é conectado ao terminal de cor verde do LED; o estado PREPARE é conectado ao terminal de cor azul e o estado SHOOT é conectado ao terminal de cor vermelha.

Para os demais LEDs, cada pino (kV, mAs e foco) foi conectado a uma cor diferente em cada um dos LEDs. Apesar de não ser extremamente necessário ao produto final, esses LEDs de visualização ajudam durante o desenvolvimento. E, como a ideia é ter apenas o botão disponível ao usuário, esses LEDs podem ser removidos posteriormente pois, afinal, o microcontrolador estará longe do usuário, junto com o gerador e o *feedback* visual é feito pela interface gráfica em si. Na Figura 20 abaixo, é possível ver a montagem do *hardware* completo.

Figura 20 – Montagem do *hardware*



Fonte: Elaboração própria.

6 RESULTADOS E DISCUSSÕES

Esta seção é dedicada à apresentação e discussão dos resultados da proposta. Nessa seção será apresentado o resultado do funcionamento do *software* e do *backend*. Também serão discutidos os testes unitários de *software* e testes de *hardware* envolvendo o funcionamento. Ademais, será feita uma análise dos requisitos de projeto após os resultados.

6.1 FUNCIONAMENTO DO SISTEMA

Os testes de funcionamento de *software* são feitos para verificar e validar o sistema desenvolvido como um todo. Nessa subseção serão apresentados os resultados pertinentes à execução principal do *backend* e do *frontend*.

Para o correto funcionamento do sistema é necessário que sejam instalados alguns pacotes essenciais da execução. Como requisitos mínimos, espera-se que o usuário tenha Python 3.8.6 instalado no Sistema Operacional Windows - verifique com o primeiro comando da lista abaixo - e com a variável de ambiente adicionada ao Path do Windows. O usuário deve inserir os seguintes códigos em um terminal aberto na pasta de projeto CIXP:

```
$ python -V
$ python -m pip install --upgrade pip
$ python -m pip install -r .\requirements.txt
```

Para o teste de *backend*, o usuário interage com o sistema através de um terminal de computador, abrindo-o na pasta “CIXP/*software*”, inserindo o seguinte comando abaixo (sem cifrão) e pressionando a tecla “Enter”. Após a execução, a finalização do exame é feita por meio da ação do usuário ao pressionar o botão de dois estágios PREPARE e SHOOT. A saída no terminal é mostrada pela Figura 21 abaixo.

```
$ python -m src.main
```

Figura 21 – Registro de execução do programa no terminal.

```
 pwsh  D:\Faculdade\212\TCC\Implementacoes\Software
> python -m src.main
=====================================================================================
                     Universidade Federal de Santa Catarina
                         CIXP - Control Interface of XP
=====================================================================================
28-Feb-22 14:48:39: INFO - Status: Iniciando conexão
28-Feb-22 14:48:39: INFO - Status: Esperando parâmetros
28-Feb-22 14:48:39: INFO - Serial communication is created (not connected yet)
28-Feb-22 14:48:39: INFO - Connecting to: COM3 at 115200 BAUD.
28-Feb-22 14:48:40: INFO - Connected to COM3 at 115200 BAUD.
28-Feb-22 14:48:42: INFO - Sending data via serial: ID request
28-Feb-22 14:48:42: INFO - Data was sent
28-Feb-22 14:48:44: INFO - Data received: Handshake-ID:CIXP
28-Feb-22 14:48:44: INFO - Handshake ID is correct: Handshake-ID:CIXP == Handshake-ID:CIXP
28-Feb-22 14:48:44: INFO - Status: Esperando o envio
28-Feb-22 14:48:44: INFO - Status: Ocupado enviando
28-Feb-22 14:48:44: INFO - Dictionary with parameters: {"parameters": {"kV": 150, "mAs": 800.0, "focus": "fine"}}
28-Feb-22 14:48:44: INFO - Sending data via serial: {"parameters": {"kV": 150, "mAs": 800.0, "focus": "fine"}}
28-Feb-22 14:48:44: INFO - Data was sent
28-Feb-22 14:48:45: DEBUG - Data received: {"parameters":{"kV":150,"mAs":800,"focus":"fine"},"ack":1}
28-Feb-22 14:48:45: INFO - ACK received
28-Feb-22 14:48:45: DEBUG - Exam has started. ACK Received
28-Feb-22 14:48:45: INFO - Status: Parâmetros enviados
28-Feb-22 14:48:45: INFO - IDLE
28-Feb-22 14:48:45: INFO - IDLE
28-Feb-22 14:48:45: INFO - IDLE
28-Feb-22 14:48:45: INFO - IDLE
28-Feb-22 14:48:45: INFO - IDLE
28-Feb-22 14:48:45: INFO - IDLE
28-Feb-22 14:48:46: INFO - IDLE
28-Feb-22 14:48:46: INFO - IDLE
28-Feb-22 14:48:46: INFO - IDLE
28-Feb-22 14:48:46: INFO - IDLE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:46: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:47: INFO - PREPARE
28-Feb-22 14:48:48: INFO - PREPARE
28-Feb-22 14:48:48: INFO - PREPARE
28-Feb-22 14:48:48: INFO - PREPARE
28-Feb-22 14:48:48: INFO - PREPARE
28-Feb-22 14:48:48: INFO - PREPARE
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:48: INFO - SHOOT
28-Feb-22 14:48:49: INFO - SHOOT
28-Feb-22 14:48:49: INFO - SHOOT
28-Feb-22 14:48:49: INFO - SHOOT
28-Feb-22 14:48:49: INFO - SHOOT
28-Feb-22 14:48:49: INFO - SHOOT
28-Feb-22 14:48:49: INFO - SHOOT
28-Feb-22 14:48:49: INFO - IDLE
28-Feb-22 14:48:49: INFO - Done
28-Feb-22 14:48:49: INFO - Disconnecting...
Disconnected.
 pwsh  D:\Faculdade\212\TCC\Implementacoes\Software
>
```

Portanto, é possível acompanhar através dos registros no terminal todos os processos de execução de um exame com parâmetros definidos ainda em main.py. O *software* inicia a execução se conectando ao dispositivo e em seguida envia uma mensagem para o mesmo que responde com a mensagem de parâmetros iguais, porém com uma *flag* de ciência de mensagem (*flag ack*).

Em seguida, o *software* mostra o estado atual do botão de disparo, já que agora o botão está liberado para tal. Assim que o usuário pressiona o botão de preparo, o estado deve continuar em PREPARE por um tempo mínimo (definido no *firmware*) e só assim poderá seguir com o disparo por um tempo máximo também definido no *firmware*. Vide “CIXP/*firmware*/CIXPM_F/include/definitions.h” ou Apêndice D para mais detalhes sobre os tempos.

6.2 TESTES UNITÁRIOS

Os testes unitários foram feitos utilizando o módulo unittest do Python. Os arquivos de testes foram postos em uma pasta separada para isolar os testes do funcionamento do código e tornar mais organizado.

Os testes unitários têm objetivo de testar cada parte de código separadamente, essa parte pode ser uma função ou classes e seus métodos. Como o desenvolvimento desse trabalho foi voltado à orientação de objeto, os testes foram escritos pensando em isolar cada classe e realizar diferentes testes de validação. Os testes são compostos com casos de teste (test Case) e cada classe possui um ou mais testes de caso, podendo ser relacionados diretamente à classe como um todo ou métodos separadamente.

Para os casos em que o método ou classe dependem de uma instância externa, como por exemplo um dispositivo ou protocolo, é utilizada a biblioteca de **Mock** de unittest. Mock permite substituir partes do sistema sob teste com objetos que imitam o comportamento dessa parte do sistema e então é possível realizar afirmações sobre esse comportamento. No caso desenvolvido, a comunicação serial com o dispositivo foi encapsulada utilizando mock, ou seja, ao invés de conectar diretamente o dispositivo e depender dele além de ter que iniciar tudo para que ele funcione, podemos simplesmente emular o comportamento de alguns métodos separadamente, como se tivéssemos utilizando o dispositivo em si. Esse conceito é importante para isolar o código testado de dependências exteriores e permitir que seja testado apenas o funcionamento do código em si.

As classes Exam e Communication utilizaram a biblioteca unittest.mock para a realização de testes. Na classe Communication todos os métodos foram testados utilizando mock para não depender da resposta de um *hardware* real e sua comunicação. Um exemplo é o método *handshake*() que utilizou mock para emular os métodos *send*()

com retorno *True* ou *False* e *read*() com o retorno do ID de *handshake*. Já a classe Exam, além de possuir alguns mock's para a realização dos testes, também possui alguns testes sem esse recurso.

Os testes consistem em: testar o retorno esperado de um método; testar o tipo de variável; testar as exceções geradas em alguns casos; testar o valor esperado de alguns atributos; comparar termos recebidos e esperados; testar. Em geral foram utilizados as assertivas: *assertEqual, assertTrue, assertFalse, assertRaises, assertLessEqual, assertIsInstance, assertIsNone.*

As subseções em sequência descrevem os testes para cada módulo/classe do sistema. No total foram 30 casos de testes aplicados inicialmente para verificar os métodos de classes e garantir funcionamento do sistema. O desenvolvimento dos casos de teste e os testes ajudaram no desenvolvimento do código na procura de falhas. Alguns métodos não estavam tão preparados para alguns cenários diferentes e, durante o desenvolvimento dos testes, foi possível perceber algumas falhas e brechas para erros, as quais foram consertadas e testadas novamente. Conforme a necessidade e aprofundamento do código, é aconselhado para trabalhos futuros aumentar a quantidade de testes e profundidade de assertivas testadas para cada vez mais refinar o código em um desenvolvimento contínuo para atingir mais qualidade.

6.2.0.1 Classe de teste Buttons

As classes de testes do módulo buttons consistem em testar os métodos da classe StageButton e Stage, respectivamente:

class TestStageButton(unittest.TestCase):

- test_init(): para testar retorno None e para testar se o estágio criado é IDLE;
- test_property(): para testar se o atributo é uma instância de Stage;
- test_setter(): para testar se o método de atribuição está correto.

class TestStage(unittest.TestCase):

- test_enum_number(): testa se os valores do enum estão corretos e em ordem.

6.2.0.2 Classe de teste Communication

Os métodos pertencentes a classe serialCom do módulo Communication foram agrupados em classes de teste de casos diferentes por conter cenários mais distindos. Abaixo são listados apenas os casos de testes.

- **class TestConnect():** testes de resposta da conexão e suas exceções;
- **class TestReconnect():** testes de resposta da reconexão
- **class TestSend():** testes de resposta de envio e suas exceções;
- **class TestRead():** testes de resposta de leitura e suas exceções;
- **class TestHandshake():** testes de *handshake* (independente de *read* e *send*)

- **class TestClose():** testes de encerramento de conexão e suas exceções.

Em cada uma dessas classes de teste de caso, foram verificados os métodos para os cenários de falha e sucesso.

6.2.0.3 Classe de teste Exam

Os testes da classe Exam foram feitos também emulando o comportamento de comunicação serial (com mock) e suas respostas.

class TestExam(unittest.TestCase):

- test_begin(): teste de retorno de sucesso dado determinados parâmetros;
- test_checkAck(): testar o método que confere que a resposta do microcontrolador contém "ack", ou seja, que a informação chegou ao destino;
- test_property(): testa a propriedade da classe quanto ao tipo de instância;
- test_updateButtonStage(): Basicamente testa a atribuição correta dos estágios do botão.

6.2.0.4 Classe de teste Generator

A classe Parameters é a classe mais importante a ser testada por possuir muitos argumentos que são cruciais para o funcionamento do programa e o comportamento incorreto pode ser danoso ao sistema. Então nessa classe de teste de caso há uma grande quantidade de testes e cada tipo de teste possui testes de validação para os diversos casos que podem ocorrer, são eles:

class TestParameters(unittest.TestCase):

- test_init(): testar o construtor da classe;
- test_type(): testa os tipos de variáveis para kV, mAs, e focus;
- test_kV_value(): testa os diferentes valores possíveis para kV;
- test_mAs_value(): testa os diferentes valores possíveis para mAs;
- test_focus_value(): testa os diferentes valores possíveis para focus;
- test_dumps(): testa o método de conversão JSON para string;

class TestGenerator(unittest.TestCase):

- test_init(): testar o construtor da classe;
- test_property(): testa a atribuição de valor à propriedade.

6.2.0.5 Classe de teste Status

Os testes para a classe Status são mais simples pelo fato da classe ser pequena. Porém ainda é necessário ter certeza que os métodos estão funcionando corretamente. Nesse sentido, há apenas uma classe de teste de caso:

class TestStatus(unittest.TestCase):

- test_setter(): testa a atribuição de valor à propriedade;
- test_description(): testa o retorno da descrição do status.

6.2.0.6 Resultados dos testes Unitários

Os testes podem ser executados separadamente ou em conjunto. Para executar os testes separados é necessário executar o seguinte comando:

```
$ python -m tests.test_buttons -v
$ python -m tests.test_communication -v
$ python -m tests.test_exam -v
$ python -m tests.test_generator -v
$ python -m tests.test_status -v
```

Para executar todos os testes da pasta em uma única vez, é necessário executar o comando `python -m unittest -v` . O resultado dos 30 testes é mostrado abaixo.

```
.
----------------------------------------------------------------------
Ran 30 tests in 0.018s

OK
```

Portanto todos os testes passaram. É importante ressaltar o constante desenvolvimento dos testes em paralelo com a aplicação para que possa ser encontrados erros durante o desenvolvimento, assim como aconteceu durante este projeto.

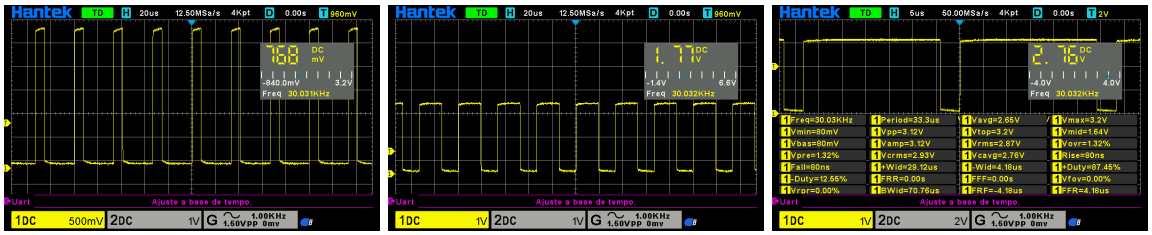
6.3 TESTES DE *HARDWARE*

Um dos objetivos dos testes de *hardware* é validar as tensões de saídas médias para os pinos de kV (18) e mAs (19) configurados como saídas PWM para os canais 0 e 2 respectivamente. Outro teste é verificar se a saída do pino 21 (Focus) está de acordo com a configuração do foco definido pelo usuário. Um terceiro teste é verificar o acionamento do botão de estágio quanto ao momento de acionamento e estágios.

6.3.0.1 Teste 1 - Teste de saída PWM para entradas kV

Os pinos 19 e GND foram conectados ao osciloscópio para medir a tensão de saída do pino de kV. Os testes foram feitos para kV com valores mínimo, intermediários e máximo. Abaixo na Figura 22 é demonstrado o resultado para os valores de 40kV, 100kV e 150kV.

Figura 22 – Testes com osciloscópio para 40, 95 e 150kV.

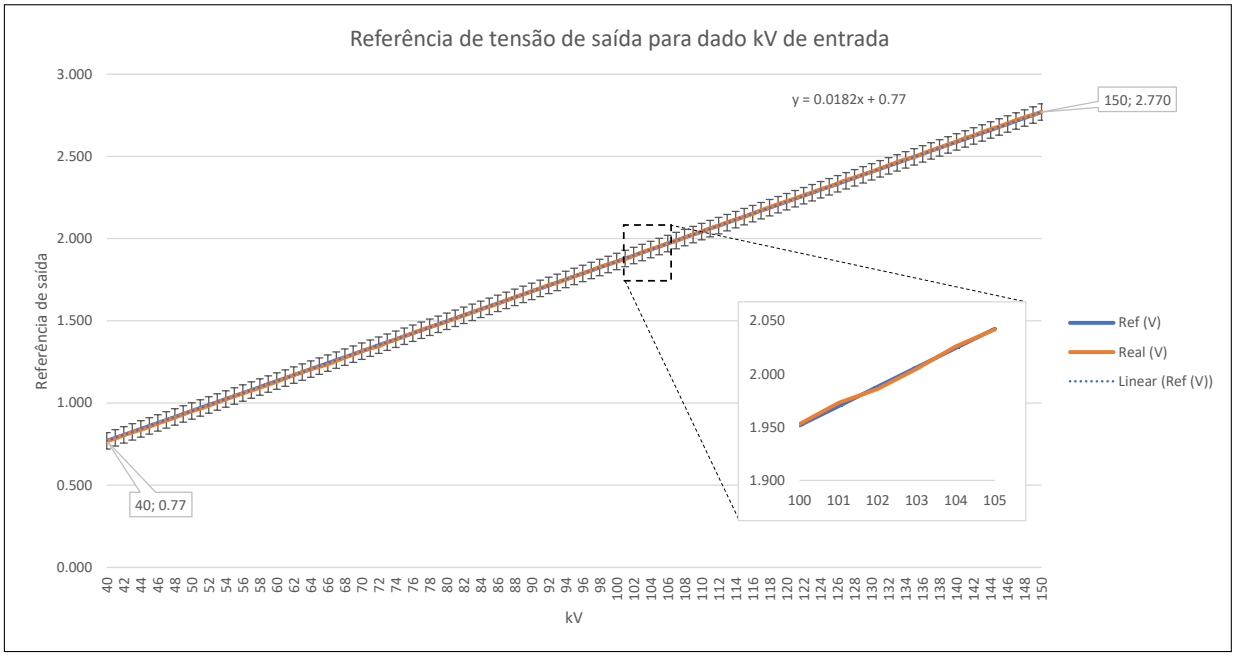


(a) 40kV. (b) 95kV (c) 150kV

Fonte: Elaboração própria.

As saídas de referência estão muito próximas às saídas estipuladas por tabela (vide Apêndice B) . O mesmo teste é feito para mais valores de kV linearmente postos considerando o mínimo 40kV (0,77V) e o máximo 150kV (2,77V). Os resultados de alguns valores, de 1 em 1 kV, podem ser vistos na Figura 23 abaixo.

Figura 23 – Resultado para entradas de kV.



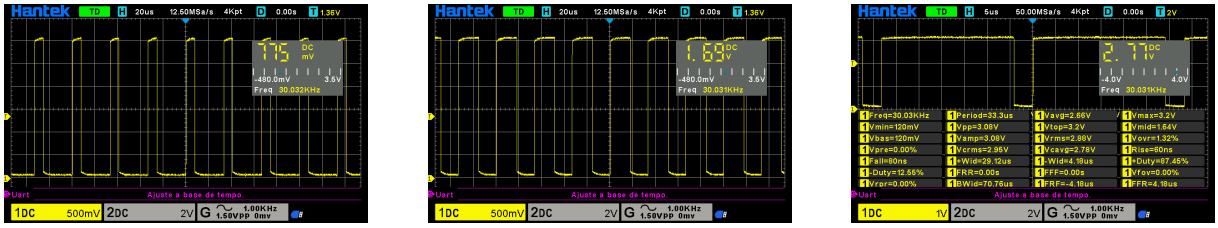
Fonte: Elaboração própria.

Os resultados comparativos entre o valor esperado e o valor atingido são bem consistentes com os resultados esperados no Apêndice B [7] possuindo erro absoluto médio de -1.07×10^{-4} e um erro relativo médio de -0.0621%. Nenhum dos valores ultrapassou o limite máximo de erro de 3% do valor, como pode ser visto nas barras de erros. Mais detalhes dos resultados em tabela do teste para entrada kV estão no Apêndice B [7].

6.3.0.2 Teste 1 - Teste de saída PWM para entradas mAs

Para as entradas de mAs, os mesmos testes aplicados para kV foram aplicados para mAs, porém com algumas diferenças. Para o teste de mAs osciloscópio foi conectado ao pino 18 do ESP32 e ao GND da protoboard. Na Figura 24 são mostrados os resultados do osciloscópio para os valores de mAs mínimo (40,0), intermediário (380,0) e máximo (800,0).

Figura 24 – Testes com osciolioscópio para 40, 380 e 800mAs.



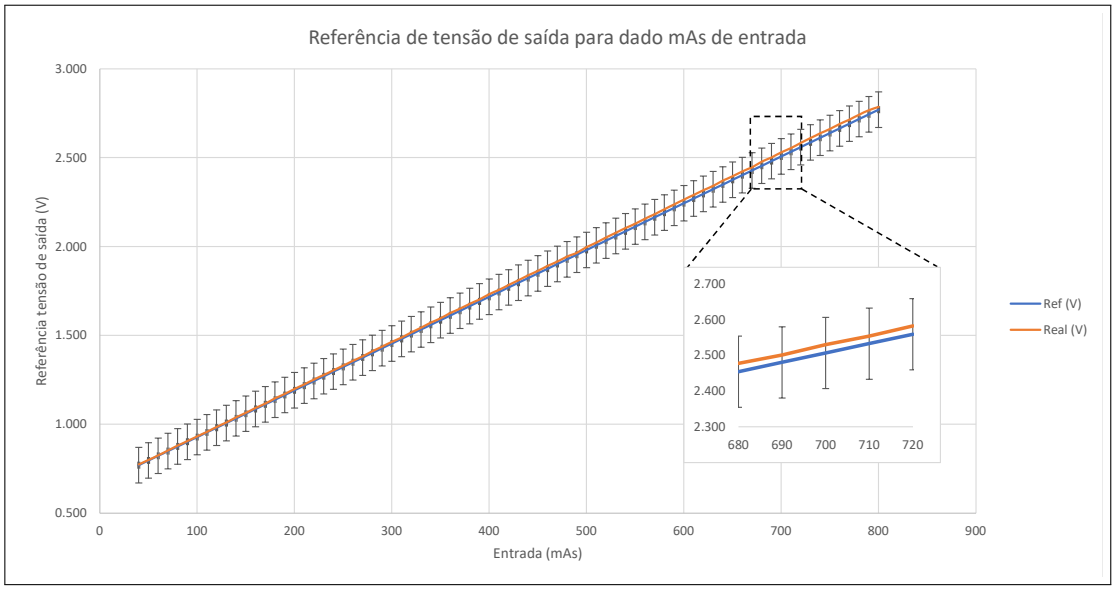
(a) 40,0mAs. (b) 380,0mAs (c) 800,0mAs

Fonte: Elaboração própria.

Em uma entrada de 40mAs é esperado valor de saída 0,77V; para uma entrada de 380mAs é esperada uma saída de 1,665V; para entrada de 800.0mAs é esperada uma saída de 2,77V.

O mesmo teste é feito também para mais valores de mAs linearmente dispostos considerando o mínimo 40mAs (esperado 0,77V) e o máximo 800mAs (esperado 2,77V). Os resultados de alguns pontos, de 10 em 10 mAs, podem ser vistos na Figura 25 abaixo.

Figura 25 – Resultado para entradas de mAs.



Fonte: Elaboração própria.

Os valores de saída de referência para mAs estão muito próximos ao teórico para todos os valores aquisitados. O erro absoluto médio é de 0,013 enquanto o erro relativo médio é de $0,7029\%$. Nenhum dos valores ultrapassou o limite máximo de erro de 3% do valor, como pode ser visto nas barras de erros. Os resultados dos testes de saídas de referência para os parâmetros de entrada de kV e mAs são apresentados com mais detalhes no Apêndice B [7].

O último teste é mais simples, caso o usuário envie “foco grosso” na seleção de foco, a saída do pino FOC_PIN deve estar em nível lógico baixo. Caso seja enviado “foco fino” o pino deve apresentar nível lógico alto. E isso é verificado através de um LED conectado ao pino e o resultado foi conforme o esperado.

6.3.1 Análise de requisitos

Nessa subseção é feita uma análise dos requisitos propostos no Capítulo 3. Na Tabela 2 é verificado requisito a requisito quanto ao cumprimento ou não dos requisitos propostos na Seção 3.

Tabela 2 – Tabela de Requisitos

Requisito	Descrição	Cumprido
RF01	Receber parâmetros de disparo do usuário: kV, mAs e foco	Sim
RF02	Tecla e indicador de liga/desliga	Sim
RF03	Enviar para o *hardware* as informações de disparo configuradas	Sim
RF04	Monitor de estados do sistema	Sim
RF05	Receber os dados seriais do computador referente ao disparo	Sim
RF06	Mapear os parâmetros de acordo com tabela de tensão de referência	Sim
RF07	Enviar um nível de tensão em cada saída analógica (uma para cada parâmetro)	Sim
RF08	Botão de disparo de raios-X de dois estados.	Sim
RNF01	Conceito *mobile first* na janela;	Sim
RNF02	Ambiente voltado para facilitar a experiência do usuário	Sim
RNF03	Envio de informações isenta de erros	Sim
RNF04	Garantia de limite de segurança dos parâmetros de disparo no *software* e *firmware*	Sim
RNF05	Precisão de duas casas decimais na leitura de saída de tensão do *hardware* para todas as saídas analógicas	Sim
RNF06	Tolerância de +-3% V nas saídas analógicas	Sim
RNF07	Sistema Operacional Windows compatível com *builds* do computador em vigor homologado (21H2)	Sim
RNF08	Processo de instalação e execução de software	Sim
RNF09	Controle de versões	Sim
RNF10	Sem acesso à terceiros ao *software*	Sim
RNF11	Informações pessoais dos usuários não podem ser vistas pelos operadores do sistema	Sim
RNF12	Controle de acesso ao *software*	Sim
RNF13	Variação de kV entre 40 e 150, passos de 1 em 1 kV	Sim
RNF14	Variação de mAs de 40 A 800	Sim
RNF15	Se o usuário cancelar o exame, resetar os parâmetros	Sim
RNF16	Se o usuário deslogar do sistema, reiniciar o mesmo	Sim
RNF17	Se o usuário desligar o sistema, deslogar o usuário e fechar programa	Sim

Fonte: Elaboração própria.

7 CONCLUSÕES

Com objetivo de desenvolver uma interface de controle de um conjunto radiológico, foram desenvolvidos códigos de *software*, *firmware* e uma interface gráfica para resolver a descentralização dos comandos em uma solução unificada.

Toda análise para desenvolver o sistema começou com o levantamento de requisitos para o sistema, os quais foram cumpridos após o desenvolvimento. Os dispositivos e ferramentas utilizadas no projeto foram suficientes para garantir que o resultado final fosse ao encontro dos requisitos propostos.

O desenvolvimento foi dividido em quatro partes: *backend* de *software*, *firmware*, Fronend de *software* e *hardware*. O desenvolvimento do *backend* de *software* foi capaz de realizar as atividades de conexão, parametrização e envio ao *firmware*. Bem como o *software*, o *firmware* foi desenvolvido com sucesso para realizar a recepção dos parâmetros provenientes do *software*, conversão dos dados em sinais de tensão e gerenciamento do botão de dois estágios.

A interface gráfica de usuário desenvolvida conseguiu cumprir os requisitos de experiência de usuário, interação com o usuário, conexão com *backend* e gerenciamento do exame.

Medidas de segurança foram realizadas para garantir o correto funcionamento do programa sob diversas condições. Também foram realizados testes unitários de *software*, testes de *backend* e testes de *hardware* para verificar e validar o desenvolvimento do protótipo.

Os resultados obtidos foram ao encontro dos resultados esperados tanto para as saídas de tensão do microcontrolador quanto para execução do programa (*backend* e *frontend*). Os resultados dos testes unitários também mostram a assertividade das classes e métodos envolvidos.

Com os resultados observados concluiu-se que o projeto desenvolvido resolveu a problemática de descentralização da interface de comando e cumpriu com sucesso os requisitos esperados para a aplicação.

Para trabalhos futuros pode-se estudar o desenvolvimento de um circuito elétrico de saída do *hardware* para estabilização e mais precisão dos resultados. Também é sugerido o estudo da auto-calibração dos níveis de tensão dos pinos de saída antes da realização do exame. Além disso, as curvas de referência de saída podem ser remodeladas conforme curvas mais exatas de um tubo de raios-X.

REFERÊNCIAS

BEISER, A. **Concepts of modern physics**. 6. ed. New York: McGraw-Hill, 2003.

BLANCHON, B. **ArduinoJson: A JSON library for embedded C++.** 2021. Disponível em: https://arduinojson.org/.

CURRY, T. S.; DOWDEY, J. E.; MURRY, R. C. **Christensen's physics of diagnostic radiology**. Philadelphia, Pennsylvania: Lippincott Williams & Wilkins, 1990.

EINSTEIN, A. Proposal of the photon concept. **American Journal of Physics**, v. 33, n. 5, p. 367–374, 1965.

FLEMING, J. A. On electric discharge between electrodes at different temperatures in air and in high vacua. **Proceedings of the Royal Society of London**, The Royal Society London, v. 47, n. 286-291, p. 118–126, 1890.

NAJARIAN, K.; SPLINTER, R. **Biomedical signal and image processing**. Boca Raton: Taylor & Francis, 2012.

PAXTON, W. F. **Thermionic electron emission properties of nitrogen-incorporated polycrystalline diamond films**. Tese (PhD Thesis - Electrical Engineering) — Vanderbilt University, Nashville, 2013.

SHUNG, K. K.; SMITH, M.; TSUI, B. M. **Principles of medical imaging**. London: Academic Press, 2012.

SOARES, J. C. d. A. **Princípios básicos de física em radiodiagnóstico**. 2. ed. São Paulo: Colégio Brasileiro de Radiologia, 2008.

SUETENS, P. **Fundamentals of medical imaging**. Cambridge, UK: Cambridge University Press, 2009.

APÊNDICE A - CÓDIGOS DE IMPLEMENTAÇÃO DO FIRMWARE

IMPLEMENTAÇÃO CLASSE BUTTON (BUTTON.CPP)

```
1
2 #include "Arduino.h"
3 #include "button.h"
4
5 Button::Button(uint8_t pin, String id) {
6   _pin = pin;
7   _id = id;
8
9   _state = false;
10  _startDebounce = false;
11  _wasPressed = false;
12
13  debounceMs = DEBOUCEMS;
14  pinMode(_pin, INPUT_PULLUP);
15 }
16
17 boolean Button::wasPressed(void) {
18     return _wasPressed;
19 }
20
21 void Button::reset(void) {
22     _wasPressed = false;
23     _startDebounce = 0;
24 }
25
26 void Button::poll(uint16_t now){
27     _state = !digitalRead(_pin);
28     if(!_state){
29         _startDebounce = 0;
30         _wasPressed = false;
31     }else{
32         if(now - _startDebounce > debounceMs) {
33             _wasPressed = true;
34         }
35     }
36 }
```

```
37
38 void Button::buttonAction(uint8_t outputPin){
39     if(wasPressed()) {
40         digitalWrite(outputPin, HIGH);
41     }else{
42         digitalWrite(outputPin, LOW);
43         reset();
44     }
45 }
46
47 void Button::setPin(uint8_t pin){
48     _pin = pin;
49 }
50
51 void Button::setId(String id){
52     _id = id;
53 }
54
55 String Button::getId(void)const{
56     return _id;
57 }
58
59 uint8_t Button::getPin(void)const{
60     return _pin;
61 }
```

IMPLEMENTAÇÃO CLASSE STAGEBUTTON (STAGEBUTTON.CPP)

```
1 #include "stagebutton.h"
2
3 using namespace stb;
4 typedef StageButton st;
5
6 st::StageButton(Button *button1, Button *button2):
7     _stage(Stage::IDLE), _enable(false),
8     b1(button1), b2(button2)
9 {
10     digitalWrite(IDLE_OUTPUT, LOW);
11     digitalWrite(PREPARE_OUTPUT, LOW);
12     digitalWrite(SHOOT_OUTPUT, LOW);
13 }
14
```

```
15 void st::setStage(Stage new_stage){
16     _stage = new_stage;
17 }
18
19 st::Stage st::getStage(void) const{
20     return _stage;
21 }
22
23 void st::update(uint16_t now){
24     if(_enable){
25         b1->poll(now);
26         b2->poll(now);
27
28         if(b1->wasPressed() == true && b2->wasPressed() == false){
29             _stage = Stage::PREPARE;
30         }
31         else if(b1->wasPressed() == false && b2->wasPressed() ==
               ↪ true){
32             if(_stage == Stage::PREPARE){
33                 _stage = Stage::SHOOT;
34             }
35         }
36         else if(b1->wasPressed() == false && b2->wasPressed() ==
               ↪ false){
37             _stage = Stage::IDLE;
38         }
39         else{
40             return;
41         }
42
43     }
44 }
45
46 bool st::isEnable(void){
47     return _enable;
48 }
49
50 void st::enable(void){
51     _enable = true;
52 }
53
```

```
54 void st::disable(void){
55     _enable = false;
56 }
57
58 void st::writeStage(void){
59     if(getStage() == st::Stage::PREPARE){
60             digitalWrite(PREPARE_OUTPUT, HIGH);
61             digitalWrite(IDLE_OUTPUT, LOW);
62         }
63         else if(getStage() == st::Stage::SHOOT){
64             digitalWrite(IDLE_OUTPUT, LOW);
65             digitalWrite(PREPARE_OUTPUT, LOW);
66             digitalWrite(SHOOT_OUTPUT, HIGH);
67
68         }
69         else{
70             digitalWrite(PREPARE_OUTPUT, LOW);
71             digitalWrite(SHOOT_OUTPUT, LOW);
72             digitalWrite(IDLE_OUTPUT, HIGH);
73     }
74 }
75
76 void st::toIdle(){
77     _stage = Stage::IDLE;
78     digitalWrite(PREPARE_OUTPUT, LOW);
79     digitalWrite(IDLE_OUTPUT, LOW);
80     digitalWrite(SHOOT_OUTPUT, LOW);
81
82 }
```

IMPLEMENTAÇÃO CLASSE MESSAGE (MESSAGE.CPP)

```
1 #include "message.h"
2
3 bool Message::receive(){
4     String incoming = Serial.readStringUntil('\n');
5
6     if(incoming == "ID request"){
7         handshake();
8         return false;
9     }
10
```

```
11      int stmsg_len = incoming.length() + 1;
12      char payload[stmsg_len];
13      incoming.toCharArray(payload, stmsg_len);
14
15      StaticJsonDocument<doc_len> doc;
16      DeserializationError error = deserializeJson(doc, incoming);
17      if (error) {
18          error_handeling(error.c_str());
19          return false;
20      }else{
21          m_doc = doc;
22          return true;
23      }
24  }
25
26  void Message::handshake(){
27      Serial.print("Handshake-ID:CIXP");
28  }
29
30  void Message::ack(){
31      StaticJsonDocument<doc_len> ack_doc;
32      JsonObject parameters = ack_doc.createNestedObject("parameters"
            ↪ );
33      parameters["kV"] = m_doc["parameters"]["kV"];
34      parameters["mAs"] = m_doc["parameters"]["mAs"];
35      parameters["focus"] = m_doc["parameters"]["focus"];
36      ack_doc["ack"] = 1;
37      String output;
38      serializeJson(ack_doc, output);
39      Serial.println(output);
40  }
41
42  void Message::updateStatus(stb::StageButton st){
43      String output = stringfy(st);
44      Serial.println(output);
45  }
46
47  String Message::stringfy(stb::StageButton st){
48      typedef stb::StageButton::Stage s;
49      switch(st.getStage()) {
50          case s::IDLE:
```

```
            return "IDLE";
        case s::PREPARE:
            return "PREPARE";
        case s::SHOOT:
            return "SHOOT";
        default :
            return "ERROR STAGE";
    }
}

void Message::error_handeling(const char *error){
    StaticJsonDocument<doc_len> doc;
    doc["reason"] = "deserialization failed:";
    doc["error"] = error;
    String output;
    serializeJson(doc, output);
    Serial.println(output);
}

StaticJsonDocument<doc_len> Message::getMessage()const{
    return m_doc;
}
```

IMPLEMENTAÇÃO CLASSE GENERATOR (GENERATOR.CPP)

```
#include "generator.h"

Generator::Generator(StaticJsonDocument<doc_len> doc){
    kV = (uint8_t)doc["parameters"]["kV"];
    mAs = (float_t)doc["parameters"]["mAs"];
    String f = doc["parameters"]["focus"];
    if (f == "thick"){
        focus = false;
    }else{
        focus = true;
    }
}

uint8_t Generator::getkV(){
    return kV;
}

```

```
18 float_t Generator::getmAs(){
19     return mAs;
20 }
21
22 bool Generator::getFocus(){
23     return focus;
24 }
```

IMPLEMENTAÇÃO CLASSE CONTROL (CONTROL.CPP)

```
1  #include "control.h"
2  #include <cmath>
3
4  Control::Control(){
5      kV_PWM = 0;
6      mAs_PWM = 0;
7      focus_output = false;
8      ledcAttachPin(KV_PIN, CHANNEL_0);
9      ledcSetup(CHANNEL_0, FREQ_HZ, RESOLUTION);
10
11
12     ledcAttachPin(MAS_PIN, CHANNEL_2);
13     ledcSetup(CHANNEL_2, FREQ_HZ, RESOLUTION);
14     pinMode(FOC_PIN, OUTPUT);
15 }
16
17 void Control::getPWM_values(Generator g){
18     kV_PWM = convertkV_PWM(g.getkV(), LO_KV, HI_KV, LO_REF_KV,
          ↪ HI_REF_KV, RESOLUTION);
19     mAs_PWM = convertmAs_PWM(g.getmAs(), LO_MAS, HI_MAS, LO_REF_MAS
          ↪ , HI_REF_MAS, RESOLUTION);
20     focus_output = g.getFocus();
21 }
22
23 /*
24 Input: variable to convert (int kV or float mAs)
25 Output: value that goes on ledcWrite function of PWM
26 */
27
28 uint16_t Control::convertkV_PWM(uint8_t input, uint8_t lo_var,
     ↪ uint8_t hi_var, float_t lo_ref, float_t hi_ref, uint8_t
     ↪ resolution){
```

```
29      uint8_t delta_var = hi_var - lo_var;
30      float_t delta_ref = hi_ref - lo_ref;
31      float_t duty_cycle;
32      float_t multimeter_error = MULTIMETER_ERROR;
33
34
35      if(resolution < 1 || resolution > 16){ return 0;}
36
37      // return 1024;
38
39      if(input < LO_KV){
40          return 0;
41      }
42
43      duty_cycle = ((input - lo_var)*(delta_ref)+ lo_ref*delta_var)
           ↪ *((hi_ref-multimeter_error)/VCC_KV_WITHOUT_LED)/(hi_ref*
           ↪ delta_var);
44      return (uint16_t)round(duty_cycle * pow(2, resolution));
45  }
46
47  uint16_t Control::convertmAs_PWM(float_t input, float_t lo_var,
       ↪ float_t hi_var,float_t lo_ref, float_t hi_ref, uint8_t
       ↪ resolution){
48      float_t delta_var = hi_var - lo_var;
49      float_t delta_ref = hi_ref - lo_ref;
50      float_t duty_cycle;
51      float_t multimeter_error = MULTIMETER_ERROR;
52
53      if(resolution < 1 || resolution > 16){ return 0;}
54
55
56      if(input < LO_MAS){
57          return 0;
58      }
59
60      duty_cycle = ((input - lo_var)*(delta_ref)+ lo_ref*delta_var)
           ↪ *((hi_ref+multimeter_error)/VCC_MAS_WITHOUT_LED)/(hi_ref*
           ↪ delta_var);
61      return (uint16_t)round(duty_cycle * pow(2, resolution));
62  }
63
```

```
64 void Control::write_kV(uint16_t pwm_value){
65     ledcWrite(CHANNEL_0, pwm_value);
66 }
67 
68 void Control::write_mAs(uint16_t pwm_value){
69     ledcWrite(CHANNEL_2, pwm_value);
70 }
71 
72 void Control::write_focus(){
73     if(focus_output == true){
74         digitalWrite(FOC_PIN, HIGH);
75     }else{
76         digitalWrite(FOC_PIN, LOW);
77     }
78 }
79 
80 void Control::writeParameters(){
81     write_mAs(mAs_PWM);
82     write_kV(kV_PWM);
83     write_focus();
84 }
85 
86 void Control::resetParameters(){
87     write_kV(0);
88     write_mAs(0);
89     focus_output = false;
90     write_focus();
91 }
```

APÊNDICE B - TABELAS DE REFERÊNCIA E RESULTADOS

Tabela 3 – Tabela de referência e testes para entrada de mAs

Entrada (mAs)	Referência (V)	Medido (V)	Erro absoluto	Erro relativo
40	0.770	0.774	0.004	0.5195%
50	0.796	0.799	0.003	0.3371%
60	0.823	0.825	0.002	0.2879%
70	0.849	0.852	0.003	0.3596%
80	0.875	0.88	0.005	0.5412%
90	0.902	0.905	0.003	0.3795%
100	0.928	0.932	0.004	0.4424%
110	0.954	0.957	0.003	0.2923%
120	0.981	0.985	0.004	0.4563%
130	1.007	1.01	0.003	0.3136%
140	1.033	1.039	0.006	0.5655%
150	1.059	1.064	0.005	0.4272%
160	1.086	1.092	0.006	0.5720%
170	1.112	1.116	0.004	0.3502%
180	1.138	1.146	0.008	0.6657%
190	1.165	1.17	0.005	0.4519%
200	1.191	1.199	0.008	0.6673%
210	1.217	1.224	0.007	0.5447%
220	1.244	1.252	0.008	0.6686%
230	1.270	1.277	0.007	0.5512%
240	1.296	1.304	0.008	0.5928%
250	1.323	1.33	0.007	0.5571%
260	1.349	1.357	0.008	0.5970%
270	1.375	1.382	0.007	0.4899%
280	1.402	1.411	0.009	0.6722%
290	1.428	1.437	0.009	0.6377%
300	1.454	1.464	0.010	0.6732%
310	1.481	1.488	0.007	0.5048%
320	1.507	1.517	0.010	0.6741%
330	1.533	1.543	0.010	0.6419%
340	1.559	1.572	0.013	0.8032%
350	1.586	1.596	0.010	0.6439%
360	1.612	1.626	0.014	0.8619%

370	1.638	1.65	0.012	0.7067%
380	1.665	1.678	0.013	0.7967%
390	1.691	1.702	0.011	0.6474%
400	1.717	1.732	0.015	0.8520%
410	1.744	1.756	0.012	0.7063%
420	1.770	1.784	0.014	0.7910%
430	1.796	1.81	0.014	0.7618%
440	1.823	1.838	0.015	0.8432%
450	1.849	1.863	0.014	0.7600%
460	1.875	1.891	0.016	0.8392%
470	1.902	1.915	0.013	0.7058%
480	1.928	1.944	0.016	0.8354%
490	1.954	1.965	0.011	0.5521%
500	1.981	1.997	0.016	0.8318%
510	2.007	2.022	0.015	0.7553%
520	2.033	2.051	0.018	0.8776%
530	2.059	2.077	0.018	0.8510%
540	2.086	2.104	0.018	0.8731%
550	2.112	2.128	0.016	0.7526%
560	2.138	2.157	0.019	0.8688%
570	2.165	2.182	0.017	0.7975%
580	2.191	2.21	0.019	0.8648%
590	2.217	2.237	0.020	0.8854%
600	2.244	2.264	0.020	0.9055%
610	2.270	2.29	0.020	0.8811%
620	2.296	2.317	0.021	0.9008%
630	2.323	2.341	0.018	0.7908%
640	2.349	2.371	0.022	0.9388%
650	2.375	2.395	0.020	0.8309%
660	2.402	2.423	0.021	0.8920%
670	2.428	2.446	0.018	0.7457%
680	2.454	2.478	0.024	0.9693%
690	2.481	2.501	0.020	0.8254%
700	2.507	2.53	0.023	0.9238%
710	2.533	2.554	0.021	0.8228%
720	2.559	2.583	0.024	0.9192%
730	2.586	2.61	0.024	0.9363%
740	2.612	2.637	0.025	0.9531%

750	2.638	2.661	0.023	0.8558%
760	2.665	2.69	0.025	0.9481%
770	2.691	2.714	0.023	0.8527%
780	2.717	2.742	0.025	0.9064%
790	2.744	2.767	0.023	0.8498%
800	2.770	2.786	0.016	0.5776%

Tabela 4 – Tabela de referência e testes para entrada de kV

Entrada (kV)	Referência (V)	Medido (V)	Erro absoluto	Erro relativo
40	0.770	0.766	-0.0040	-0.5195%
41	0.788	0.785	-0.0032	-0.4035%
42	0.806	0.804	-0.0024	-0.2927%
43	0.825	0.823	-0.0015	-0.1868%
44	0.843	0.837	-0.0057	-0.6788%
45	0.861	0.856	-0.0049	-0.5692%
46	0.879	0.876	-0.0031	-0.3504%
47	0.897	0.893	-0.0043	-0.4748%
48	0.915	0.911	-0.0044	-0.4850%
49	0.934	0.932	-0.0016	-0.1735%
50	0.952	0.95	-0.0018	-0.1891%
51	0.970	0.965	-0.0050	-0.5134%
52	0.988	0.983	-0.0052	-0.5222%
53	1.006	1.003	-0.0033	-0.3319%
54	1.025	1.022	-0.0025	-0.2460%
55	1.043	1.041	-0.0017	-0.1630%
56	1.061	1.059	-0.0019	-0.1772%
57	1.079	1.076	-0.0031	-0.2836%
58	1.097	1.093	-0.0042	-0.3864%
59	1.115	1.111	-0.0044	-0.3963%
60	1.134	1.13	-0.0036	-0.3176%
61	1.152	1.15	-0.0018	-0.1545%
62	1.170	1.17	0.0000	0.0034%
63	1.188	1.188	-0.0001	-0.0118%
64	1.206	1.205	-0.0013	-0.1094%
65	1.225	1.22	-0.0045	-0.3675%
66	1.243	1.235	-0.0077	-0.6180%
67	1.261	1.256	-0.0049	-0.3855%
68	1.279	1.277	-0.0020	-0.1595%

69	1.297	1.294	-0.0032	-0.2482%
70	1.315	1.314	-0.0014	-0.1064%
71	1.334	1.332	-0.0016	-0.1185%
72	1.352	1.345	-0.0068	-0.5001%
73	1.370	1.369	-0.0009	-0.0686%
74	1.388	1.386	-0.0021	-0.1527%
75	1.406	1.405	-0.0013	-0.0924%
76	1.424	1.425	0.0005	0.0365%
77	1.443	1.444	0.0013	0.0929%
78	1.461	1.462	0.0012	0.0794%
79	1.479	1.477	-0.0020	-0.1366%
80	1.497	1.495	-0.0022	-0.1469%
81	1.515	1.514	-0.0014	-0.0911%
82	1.534	1.532	-0.0016	-0.1017%
83	1.552	1.551	-0.0007	-0.0477%
84	1.570	1.571	0.0011	0.0688%
85	1.588	1.586	-0.0021	-0.1322%
86	1.606	1.604	-0.0023	-0.1419%
87	1.624	1.625	0.0005	0.0332%
88	1.643	1.642	-0.0006	-0.0390%
89	1.661	1.661	0.0002	0.0108%
90	1.679	1.679	0.0000	0.0000%
91	1.697	1.697	-0.0002	-0.0106%
92	1.715	1.717	0.0016	0.0956%
93	1.734	1.731	-0.0025	-0.1465%
94	1.752	1.751	-0.0007	-0.0411%
95	1.770	1.769	-0.0009	-0.0509%
96	1.788	1.789	0.0009	0.0515%
97	1.806	1.806	-0.0003	-0.0144%
98	1.824	1.827	0.0026	0.1403%
99	1.843	1.844	0.0014	0.0749%
100	1.861	1.86	-0.0008	-0.0430%
101	1.879	1.875	-0.0040	-0.2118%
102	1.897	1.896	-0.0012	-0.0611%
103	1.915	1.917	0.0017	0.0867%
104	1.934	1.935	0.0015	0.0765%
105	1.952	1.953	0.0013	0.0666%
106	1.970	1.973	0.0031	0.1584%

107	1.988	1.986	-0.0021	-0.1036%
108	2.006	2.005	-0.0012	-0.0618%
109	2.024	2.026	0.0016	0.0780%
110	2.043	2.042	-0.0006	-0.0294%
111	2.061	2.062	0.0012	0.0592%
112	2.079	2.08	0.0010	0.0500%
113	2.097	2.099	0.0019	0.0887%
114	2.115	2.117	0.0017	0.0794%
115	2.134	2.132	-0.0015	-0.0703%
116	2.152	2.152	0.0003	0.0149%
117	2.170	2.172	0.0021	0.0986%
118	2.188	2.191	0.0030	0.1353%
119	2.206	2.21	0.0038	0.1713%
120	2.224	2.227	0.0026	0.1169%
121	2.243	2.244	0.0014	0.0633%
122	2.261	2.264	0.0032	0.1433%
123	2.279	2.281	0.0021	0.0904%
124	2.297	2.3	0.0029	0.1254%
125	2.315	2.317	0.0017	0.0734%
126	2.333	2.335	0.0015	0.0651%
127	2.352	2.357	0.0053	0.2271%
128	2.370	2.372	0.0022	0.0911%
129	2.388	2.39	0.0020	0.0829%
130	2.406	2.406	-0.0002	-0.0083%
131	2.424	2.427	0.0026	0.1081%
132	2.443	2.446	0.0034	0.1408%
133	2.461	2.466	0.0053	0.2138%
134	2.479	2.483	0.0041	0.1646%
135	2.497	2.496	-0.0011	-0.0441%
136	2.515	2.517	0.0017	0.0684%
137	2.533	2.536	0.0025	0.1003%
138	2.552	2.554	0.0024	0.0925%
139	2.570	2.573	0.0032	0.1237%
140	2.588	2.591	0.0030	0.1159%
141	2.606	2.612	0.0058	0.2233%
142	2.624	2.627	0.0026	0.1006%
143	2.643	2.647	0.0045	0.1688%
144	2.661	2.666	0.0053	0.1984%

145	2.679	2.682	0.0031	0.1157%
146	2.697	2.702	0.0049	0.1824%
147	2.715	2.721	0.0057	0.2114%
148	2.733	2.739	0.0056	0.2034%
149	2.752	2.754	0.0024	0.0865%
150	2.770	2.771	0.0012	0.0433%

APÊNDICE C - ÁRVORE DE ARQUIVOS DO PROJETO

```
CIXP
├── Software
│   ├── app
│   │   ├── img/
│   │   ├── cixp.py
│   │   └── qt-files/
│   ├── lib
│   │   ├── __init__.py
│   │   ├── buttons.py
│   │   ├── communication.py
│   │   ├── exam.py
│   │   ├── generator.py
│   │   └── status.py
│   ├── src
│   │   └── main.py
│   └── test
│       ├── __init__.py
│       ├── test_buttons.py
│       ├── test_communication.py
│       ├── test_exam.py
│       ├── test_generator.py
│       └── test_status.py
├── Firmware
│   └── CIXPM_F
│       ├── src
│       │   ├── button.cpp
│       │   ├── control.cpp
│       │   ├── generator.cpp
│       │   ├── main.cpp
│       │   ├── message.cpp
│       │   └── stagebutton.cpp
│       └── include
│           ├── button.h
│           ├── control.h
│           ├── definitions.h
│           ├── generator.h
│           ├── message.h
```

```
38 │                   ├── pinout.h
39 │                   └── stagebutton.h
40 ├── README.md
41 └── requirements.txt
```

Listing 1 – Árvore dos projeto

APÊNDICE D - ARQUIVOS AUXILIARES

“DEFINITIONS.H”

```
#ifndef DEFINITIONS_H
#define DEFINTIONS_H

#include "Arduino.h"

#define VCC_MAS_WITHOUT_LED 3.15//calibrado
#define VCC_MAS_WITH_LED    3.008//calibrado
#define VCC_KV_WITHOUT_LED  3.15 //calibrado
#define VCC_KV_WITH_LED     3.12 //calibrado
#define FREQ_HZ             30000 // Hz
#define CHANNEL_0           0 // kV CHANNEL
#define CHANNEL_2           2 // mAs CHANNEL
#define HI_KV               150 // kV
#define LO_KV               40  // kV
#define RESOLUTION          10 // bits
#define HI_REF_KV           2.81
#define LO_REF_KV           0.73
#define HI_MAS              800.0 // mAs
#define LO_MAS              40.0  // mAs
#define HI_REF_MAS          2.81
#define LO_REF_MAS          0.73
#define MULTIMETER_ERROR    0.0
#define SHOOT_MAX_MS        1000
#define PREPARE_MIN_MS      2000

#endif
```

“PINOUT.H”

```
#ifndef PINOUT_H
#define PINOUT_H

#include <Arduino.h>

#define IDLE_OUTPUT     5
```

```
#define PREPARE_OUTPUT  2
#define SHOOT_OUTPUT    4
#define MAS_PIN         18
#define KV_PIN          19
#define FOC_PIN         21
#define STAGE1          22
#define STAGE2          23

#endif
```

ANEXO A - REFERÊNCIA DE TENSÃO PARA DETERMINADO KV E MA

KV	REF (V)
40	0,770
41	0,790
42	0,805
44	0,830
46	0,890
48	0,940
50	0,980
52	1,015
55	1,070
57	1,105
60	1,190
63	1,225
66	1,285
70	1,380
73	1,440
77	1,520

KV	REF (V)
81	1,595
85	1,680
90	1,775
96	1,890
102	1,990
109	2,135
117	2,295
125	2,440
129	2,510
133	2,590
137	2,640
141	2,680
145	2,730
150	2,770

Figura 26 – Referência de tensão para determinado kV