

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA
CATARINA – CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA**

RAFAEL FERNANDO REIS

**SISTEMA DE INTEGRAÇÃO CONTÍNUA E AUTOMATIZAÇÃO DE
TESTES VOLTADOS AO DESENVOLVIMENTO DE PRODUTOS
ELETRÔNICOS EMBARCADOS**

FLORIANÓPOLIS, 2020.

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA
CATARINA – CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ELETRÔNICA INDUSTRIAL**

RAFAEL FERNANDO REIS

**SISTEMA DE INTEGRAÇÃO CONTÍNUA E AUTOMATIZAÇÃO DE
TESTES VOLTADOS AO DESENVOLVIMENTO DE PRODUTOS
ELETRÔNICOS EMBARCADOS**

Trabalho de Conclusão de Curso submetido ao Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina como parte dos requisitos para obtenção do título de Bacharel em Engenharia Eletrônica.

Orientador:
Prof. Me. Hugo Marcondes

FLORIANÓPOLIS, 2020.

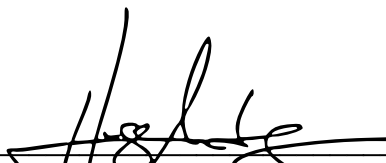
**SISTEMA DE INTEGRAÇÃO CONTÍNUA E AUTOMATIZAÇÃO DE
TESTES VOLTADOS AO DESENVOLVIMENTO DE PRODUTOS
ELETRÔNICOS EMBARCADOS**

RAFAEL FERNANDO REIS

Este trabalho foi julgado adequado para obtenção do título de Bacharel em Engenharia Eletrônica e aprovado na sua forma final pela banca examinadora do Curso Superior de Engenharia Eletrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 26 de Outubro, 2020.

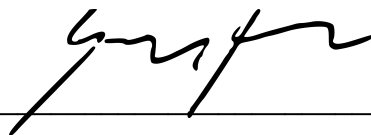
Banca Examinadora:



Hugo Marcondes, Me.

Pedro Giassi Jr.

Pedro Giassi Junior, Dr.



Samir Bonho, M.Eng.

RESUMO

Este trabalho de conclusão de curso busca pesquisar soluções de sistemas de integração contínua focadas em atender as necessidades de uma equipe de desenvolvimento de produtos embarcados, e implantá-las em um projeto executado pela mesma. Para esse fim, foi estudado o processo de desenvolvimento de sistemas embarcados, os conceitos de metodologia que utilizam a integração contínua, possíveis ferramentas para a automatização do processo, assim como os tipos de testes a serem realizados durante o desenvolvimento. O sistema busca trazer agilidade e robustez para a equipe de desenvolvimento por meio da automatização de testes conjuntos de software e hardware. Para isso, foram utilizadas as ferramentas: Bitbucket para controle de versão, o servidor de automatização Jenkins para especificações e o disparo dos testes e o LAVA para gerenciamento da execução e validação dos mesmos. Por fim, o sistema proposto foi implantado ao processo de desenvolvimento de um sistema embarcado. Este trabalho observou que a aplicação da integração contínua neste projeto contribuiu para otimizar o fluxo de trabalho da equipe de desenvolvimento.

Palavras-chave: Integração Contínua. Sistemas Embarcados. Automação. Testes. Desenvolvimento.

ABSTRACT

This end of course work seeks to research solutions for continuous integration systems focused on meeting the needs of an embedded product development team, and implementing them in a project carried out by the same. To that end, the process of developing embedded systems was studied, the concepts of methodology that use continuous integration, possible tools for the automation of the process, as well as the types of tests to be performed during development. The system seeks to bring agility and robustness to the development team through the automation of joint tests of software and hardware. For this, the used tools were: Bitbucket for version control, Jenkins automation server for specifications and the launching of tests and LAVA for management of their execution and validation. Finally, the proposed system was implemented in the process of developing an embedded system. This work noted that the application of continuous integration in this project contributed to optimize the workflow of the development team.

Keywords: Continuous integration. Embedded Systems. Automation. Tests. Development.

LISTA DE FIGURAS

Figura 1 - Etapas da integração contínua	12
Figura 2 - Estrutura interna do LAVA	15
Figura 3 - Estrutura projetada da integração contínua	19
Figura 4 - Repositório online Bitbucket.....	20
Figura 5 - Interface Jenkins	21
Figura 6 - Configuração de pipeline do Jenkins	22
Figura 7 - Exemplos da linguagem declarativa de pipeline do Jenkins	23
Figura 8 - Aba de configuração de trigger do Jenkins	24
Figura 9 - Aba de Webhooks do Bitbucket	24
Figura 10 - Interface web do LAVA	25
Figura 11 - Placa de desenvolvimento STM32H743ZI	26
Figura 12 - Resultado dos testes na interface web	27
Figura 13 - Placa de relés	29
Figura 14 - Estrutura de controle dos DUTs.....	29
Figura 15 - Armário de teste.....	30
Figura 16 - Placa de controle	31
Figura 17 - Fluxograma simplificado da CLI	32

LISTA DE ABREVIATURAS E SIGLAS

CERTI - Centros de Referência em Tecnologias Inovadoras

CLI - Command Line Interface

DUT – Device Under Test

HTTP - Hypertext Transfer Protocol

I2C - Inter-Integrated Circuit

JRE - Java Runtime Environment

LAVA - Linaro Automation and Validation Architecture

OpenOCD - Open On-Chip Debugger

PDU - Power Distribution Unit

RPC - Remote Procedure Call

SPI - Single Peripheral Interface

XML - Extensible Markup Language

YMAL - YAML Ain't Markup Language

SUMÁRIO

1	INTRODUÇÃO	7
1.1	Definição do Problema	7
1.2	Justificativa	8
1.3	Objetivo Geral.....	8
1.4	Objetivos Específicos	9
2	FUNDAMENTAÇÃO TEÓRICA	10
2.1	Projetos de sistemas embarcados	10
2.2	Integração contínua	11
2.3	Ferramentas de integração contínua	13
2.4	Tipos de teste para embarcados	17
2.4.1	Testes unitários.....	17
2.4.2	Testes de integração.....	17
2.4.3	Testes de estresse.....	18
3	DESENVOLVIMENTO	19
3.1	Repositório de arquivos online e Jenkins	20
3.2	Automatização de testes	25
3.2.1	Unidade de controle de acionamento e teste	27
4	APRESENTAÇÃO DOS RESULTADOS	30
4.1	Placa de controle desenvolvida.....	31
4.2	Erros detectados.....	33
4.2.1	Erros no build.....	33
4.2.2	Erros nos testes unitários.....	33
4.2.3	Erros de gravação de binário	34
4.2.4	Erros nos testes de estresse.....	35
5	CONCLUSÃO	36
5.1	Sugestões de trabalhos futuros	36
	REFERÊNCIAS	37
	APÊNDICES	38
	APÊNDICE A – Script de criação de arquivo de teste LAVA.....	39
	APÊNDICE B – Script de envio de arquivo de teste LAVA	41
	APÊNDICE C – CLI desenvolvida	43
	ANEXO	52
	ANEXO A – Arquivo DeviceType da placa STM32H743ZI.....	53
	ANEXO B – Arquivo Device da placa STM32H743ZI.....	55
	ANEXO C – Arquivo Health Check da placa STM32H743ZI	56

1 INTRODUÇÃO

O mercado de desenvolvimento dos mais diversos tipos de produtos eletrônicos cresce a cada dia, no qual tecnologias se tornam obsoletas com extrema rapidez. Em 2019, o tamanho do mercado de sistemas embarcados excedeu 100 bilhões de dólares americanos (WADHWANI). Somado ao crescimento exponencial da complexidade destes produtos, a necessidade por profissionais cada vez mais qualificados e atualizados em suas respectivas áreas aumenta. Tais profissionais necessitam ter conhecimento das ferramentas e metodologias de desenvolvimento adequadas, a fim de acelerar ao máximo o processo de criação e por fim a entrada do produto no mercado.

Similarmente ao avanço das tecnologias nos sistemas embarcados, as ferramentas e métodos necessários para o desenvolvimento também se adaptaram para se adequarem aos novos requisitos. Entre elas, a metodologia ágil da integração contínua se destaca por possibilitar um melhor fluxo de desenvolvimento além da entrega de projetos robustos, testados e de qualidade. Com a evolução dessas ferramentas, também foi possível a inclusão da área de hardware nesta metodologia, trazendo maior agilidade para o desenvolvimento de sistemas embarcados como um todo.

1.1 Definição do Problema

Devido à natureza das atividades elaboradas no processo de desenvolvimento de um sistema embarcado e a complexidade dos projetos, a quantidade de desenvolvedores envolvidos pode ser elevada, fator que dificulta ainda mais a organização das atividades de um projeto. Ademais, tarefas periódicas como manutenção e validação de implementações do software, quando realizadas manualmente, estão sujeitas a erros humanos que podem causar diferentes problemas para o fluxo de desenvolvimento. Por esse motivo, o desconhecimento de ferramentas e métodos especializados, como a integração contínua, pode resultar em equipamentos frágeis e instáveis, além de poder aumentar consideravelmente o tempo

necessário para finalização do produto. Com a problemática definida, se faz possível levantar as seguintes perguntas:

- a) É possível melhorar o fluxo de desenvolvimento de um sistema embarcado a partir da implementação da metodologia da integração contínua?
- b) A implementação da automatização de validação de software reduz a quantidade de erros durante o desenvolvimento?
- c) A implementação da integração contínua diminui o tempo de desenvolvimento de um sistema embarcado?

1.2 Justificativa

Devido ao mercado extremamente competitivo de eletrônicos, ferramentas e metodologias de desenvolvimento que agilizam o processo de validação e automatizam os testes dos equipamentos se fazem cruciais. Garantia de funcionamento e qualidade de um produto eletrônico é fundamental tanto para a segurança e satisfação dos usuários que os utilizam quanto para a venda do mesmo.

Em vista disso, o desenvolvimento de uma arquitetura ou sistema de fácil configuração que utilize uma metodologia de integração contínua e ferramentas especializadas, pode ser um poderoso recurso para a área industrial de desenvolvimento de produtos embarcados.

1.3 Objetivo Geral

Este trabalho de conclusão de curso busca desenvolver um sistema de integração contínua com foco na automatização de testes de firmware e hardware, englobando ferramentas utilizadas da área de desenvolvimento.

1.4 Objetivos Específicos

De modo a atender os objetivos gerais deste trabalho, foram definidos os seguintes objetivos específicos:

- a) Realizar o estudo do conceito de integração contínua e as possíveis ferramentas para sua implementação.
- b) Implementar o sistema de integração contínua para otimizar o fluxo de desenvolvimento de produtos embarcados.
- c) Avaliar a implementação do sistema desenvolvido para um ambiente de desenvolvimento de embarcados.
- d) Avaliar os resultados e erros obtidos a partir da utilização do sistema.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão abordados os principais temas a serem estudados para o desenvolvimento do sistema implementado. A partir do aprofundamento no estudo dos métodos e procedimentos para o desenvolvimento de sistemas embarcados, é possível definir as problemáticas e possíveis soluções a fim de agilizar o processo como um todo. Com o mesmo intuito, o estudo da teoria por traz da integração contínua também se faz necessária para estruturação do projeto. Outro tópico que se faz fundamental é o conhecimento das ferramentas disponíveis para auxiliar a construção do sistema, que devido a sua natureza, podem necessitar da interação de múltiplas ferramentas especializadas atuando paralelamente. Por fim, a área de testes é outro tema a ser explorado, tópico crucial para estabilidade e garantia de funcionamento do sistema completo.

2.1 Projetos de sistemas embarcados

Atualmente, sistemas embarcados estão presentes em inúmeras áreas da sociedade. Áreas como a indústria automotiva, segurança, área médica e muitas outras vem utilizando a tecnologia crescente de produtos embarcados para aperfeiçoar seus respectivos campos. Em comum a todas estas áreas da ciência está a dependência por produtos embarcados confiáveis, disponíveis, seguros e de fácil manutenção.

A fim de atender aos requisitos impostos para cada um dos equipamentos desenvolvidos, o desenvolvimento de sistemas embarcados apresenta grandes desafios tecnológicos. Eles podem ser definidos por:

- 1) Sistemas embarcados devem ser realmente confiáveis.[...]
- 2) Devido a metas de eficiência, os projetos de software não podem ser feitos independentemente do hardware utilizado.[...]
- 3) Sistemas embarcados devem atender a diversos requisitos não funcionais como restrições de tempo real, eficiência de energia e requisitos de confiabilidade.[...]
- 4) Sistemas reais são profundamente concorrentes.[...]
- 5) Linguagens de programação sequencial tradicionais não são a melhor maneira de descrever sistemas simultâneos e cronometrados. (MARWEDEL, 2011, p. 10, tradução nossa).

O projeto de sistemas embarcados é uma tarefa complexa com diversos conhecimentos atrelados. Com o intuito de facilitar o desenvolvimento, muitas vezes é

realizado a quebra do projeto em tarefas menores. Tais tarefas devem ser realizadas em sequência e em alguns casos ser repetidas. O design do sistema é iniciado a partir de formulações de especificações pelos desenvolvedores. Nesta etapa, a partir de interações de design do projeto, as aplicações do sistema são mapeadas. A validação do design do sistema deve atender diversos requisitos como: performance, confiabilidade, consumo energético, manufaturabilidade entre outras. Além disso, devido à importância da eficiência de produtos embarcados, são fundamentais as constantes otimizações do sistema (MARWEDEL, 2011).

A confiabilidade de um sistema embarcado é uma característica que, se possível, deve ser analisada a todo o momento durante a etapa de desenvolvimento. Para este fim, a utilização de ferramentas especializadas pode ser uma grande aliada.

2.2 Integração contínua

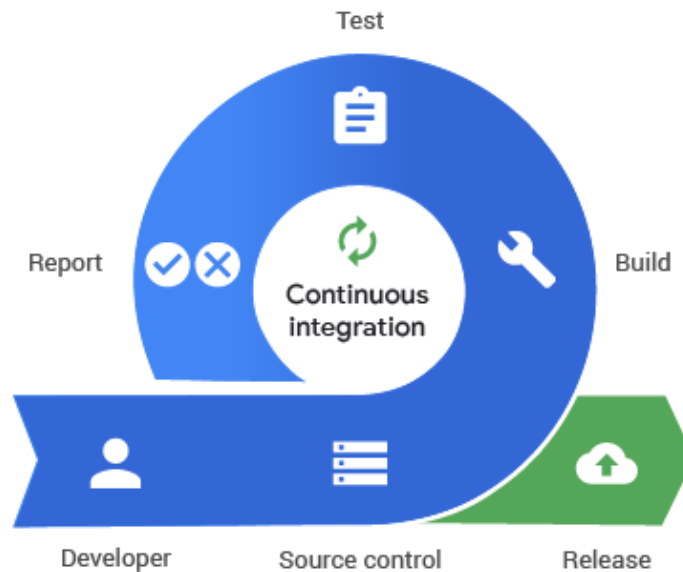
Em desenvolvimentos de software mais tradicionais, a integração de diferentes partes de um projeto são realizadas após a implementação completa dos mesmos e muitas vezes de forma isolada. Devido principalmente ao aumento na complexidade dos projetos, surgiu a necessidade por uma maior frequência de integrações entre partes dos projetos durante a etapa de desenvolvimento, sendo esta metodologia chamada de integração contínua.

A definição de integração contínua pode ser dada por:

A integração contínua é uma prática de desenvolvimento de software em que os membros de uma equipe integram seu trabalho com frequência, geralmente cada pessoa integra pelo menos diariamente - levando a várias integrações por dia. Cada integração é verificada por um build automatizado (incluindo teste) para detectar erros de integração o mais rápido possível. Muitas equipes descobrem que esta abordagem leva a reduções significativas de problemas de integração e permite que uma equipe desenvolva software coeso mais rapidamente. (FOWLER, 2006, tradução nossa)

Para desenvolvimento do sistema proposto o entendimento das etapas da metodologia ágil de integração contínua são fundamentais. A figura 1 apresenta as suas principais etapas no desenvolvimento de software.

Figura 1- Etapas da integração contínua



Fonte: Google Cloud Website (2020).

Como se pode observar, o processo se inicializa com o desenvolvedor do projeto realizando uma modificação no código do projeto e o enviando para um repositório organizado por versões. Antes de este código ser adicionado à versão final, muitas vezes chamadas de *release*, ele passa por três estágios básicos: *Build*, teste e geração de relatório. Dependendo do tipo de projeto desenvolvido, a ação realizada em cada um dos passos pode variar relativamente.

O passo de *Build* consiste em compilar o código fonte da aplicação do projeto a fim de gerar um arquivo executável. Em produtos embarcados esse processo muitas vezes agrega também a geração da imagem binária a ser utilizada para gravação no hardware do projeto. Em caso de sucesso, o processo segue para o estágio de teste.

O estágio de teste pode ser executado de diferentes formas e vai depender diretamente do tipo do projeto sendo realizado. Nesse estágio, alterações realizadas no código fonte devem ser testadas extensivamente com a utilização de testes relevantes para o produto. Em produtos embarcados, no qual existe a possibilidade do hardware do projeto também estar em desenvolvimento, esse estágio se torna ainda mais crítico no qual os testes realizados devem tanto validar o firmware quanto o hardware.

O relatório da integração contínua é a parte final do processo antes da inclusão das novas mudanças ao projeto. A partir dos resultados obtidos do estágio de

build e teste o sistema deve autorizar ou negar a inclusão das mudanças apontando os motivos no caso de falha.

Uma das grandes vantagens da utilização da integração contínua em projetos de desenvolvimento é o aumento da produtividade do desenvolvedor, que devido a grande automatização do processo, fica liberto de realizar diversas tarefas manuais repetidas durante a execução do projeto. Ademais, com todas as etapas e procedimentos concluídos, existe uma grande garantia de funcionamento da versão do código fonte a todo o momento, facilitando a investigação de possíveis *bugs* no projeto e retrabalhos devido à falta de testes (DUVAL, 2007).

2.3 Ferramentas de integração contínua

Como relatado previamente, no mercado atual existem diversas ferramentas para auxiliar a aplicação dos conceitos de integração contínua nos mais diversos tipos de projetos. Devido a grande aplicação na área de desenvolvimento de *softwares*, grande parte destas ferramentas busca auxiliar preferencialmente este gênero de projetos. Entretanto, devido à pluralidade de projetos da própria área de *software*, muitas vezes essas ferramentas são criadas com o intuito que elas sejam facilmente adaptadas para atender inúmeros tipos de projetos diferentes. Dessa maneira, tais ferramentas não somente conseguem atender a maior quantidade possível de desenvolvedores, mas também satisfazem os requisitos necessários para uma gama de projetos de outras áreas especializadas, como é o caso dos sistemas embarcados.

O servidor de automatização de processos Jenkins é uma dessas ferramentas. Disponibilizado no início do ano de 2005, o Jenkins é um poderoso aliado de milhares de desenvolvedores ao redor do mundo, aplicando aos projetos os conceitos de integração contínua por meio de uma interface simples, intuitiva, prática e com uma baixa curva de aprendizado (SMART, 2011).

Sendo um projeto de código aberto, o Jenkins se destaca devido a sua ampla biblioteca de *plugins* criados pelos próprios usuários e disponibilizados de forma totalmente gratuita. Devido a sua grande comunidade de desenvolvedores que utilizam a ferramenta, tais *plugins* são testados e revisados constantemente, trazendo robustez e confiança aos usuários do servidor.

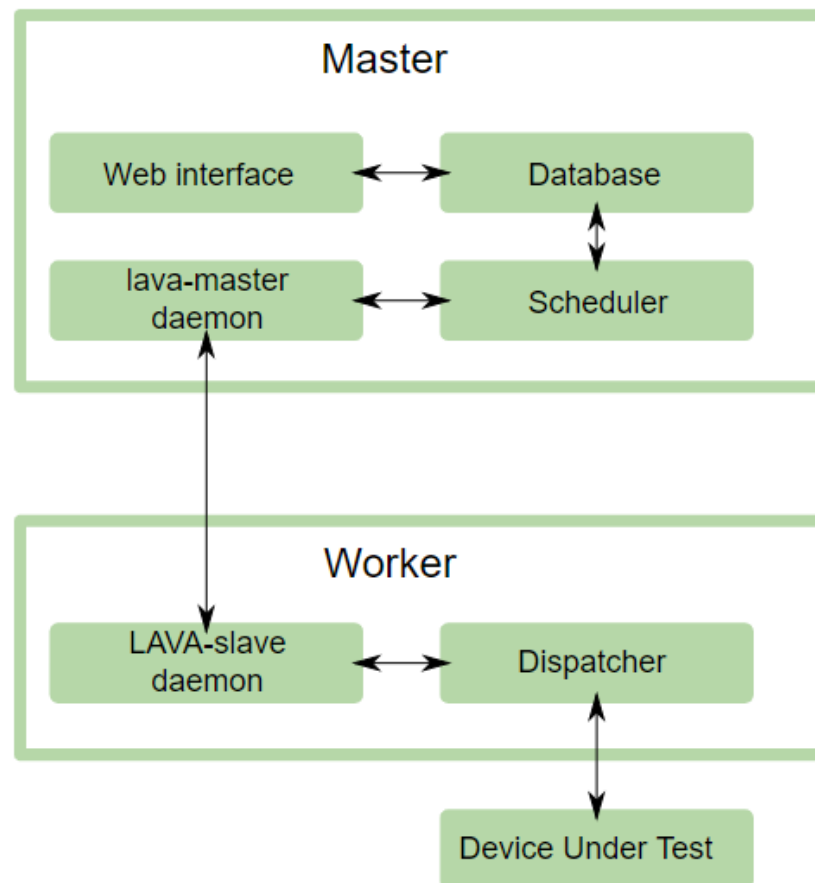
Esta ferramenta de integração contínua funciona a partir de *jobs* configuráveis na sua própria interface de usuário e permite a comunicação com diversas outras ferramentas. As ações realizadas nesses *jobs* são extremamente customizáveis, sendo o processo de build o mais comum dos casos, no qual o registro dos resultados são encontrados na interface de forma clara e simples.

A partir da documentação do Jenkins encontram-se três principais formas de instalação: Utilizando pacotes nativos do sistema operacional, executando a partir do JRE (*Java Runtime Environment*) ou utilizando em conjunto com a ferramenta Docker (JENKINS, 2020). Usado largamente no ambiente de desenvolvimento de softwares, o Docker não é uma ferramenta desenvolvida especificamente para a utilização em sistemas de integração contínua, entretanto, essa inteligente ferramenta pode trazer diversas características positivas para o sistema. A ferramenta tem como finalidade enclausurar e executar uma ou mais aplicações e suas dependências em um *container* virtual, ambiente isolado de execução da aplicação (DOCKER, 2020). Tal processo provê a portabilidade da aplicação para diferentes plataformas sem a preocupação do usuário quanto à questão de compatibilidade entre a aplicação executada dentro do container e o sistema executando o Docker. Utilizando em conjunto com o Jenkins, por exemplo, permite que o processo de instalação, configuração e execução da ferramenta de integração contínua seja enclausurado em um *container* e facilmente movido entre servidores que utilizam diferentes sistemas operacionais.

Outra ferramenta essencial para projetos de desenvolvimento de softwares são os repositórios de arquivos e controladores de versão. Tem como principal função armazenar de forma segura e organizada todos os arquivos do projeto, além de registrar todas as mudanças realizadas nos arquivos e versões existentes. Para esse fim, o sistema utilizado por muitas empresas de desenvolvimento é o GIT. Lançado em 2005 por Linus Torvalds, o sistema possui código aberto e é empregado como estrutura base por diversos repositórios web de controle de versão como: GitHub, Bitbucket, GitLab e muitos outros. É importante ressaltar a necessidade de comunicação entre o repositório de arquivos e a ferramenta de integração contínua, já que todo o processo é inicializado a partir de modificações no próprio repositório e a etapa de *build* depende completamente dos arquivos providos. No caso do Jenkins, muitas vezes esta comunicação pode ser feita através de *plugins*.

Por fim, com o intuito de administrar os testes e prover comunicação direta com as placas utilizadas em um projeto embarcado, novas ferramentas especializadas podem ser utilizadas. Esse é o caso do LAVA (*Linaro Automation and Validation Architecture*). Esta arquitetura tem como sua especialidade administrar e enfileirar os testes nas placas conectadas ao sistema, além de automatizar três importantes passos no processo de teste dos novos firmwares: *Deploy*, *Boot* e *Monitor*. A figura 2 mostra a estrutura interna do LAVA.

Figura 2 - Estrutura interna do LAVA



Fonte: LAVA Website (2020).

Como pode ser observada na figura 2, o LAVA é separado entre *master* e *worker*. O *master* da ferramenta conta com uma interface web para utilização do usuário com informações dos testes realizados e placas conectadas. Todas essas informações são armazenadas na data-base de modo a permitir acesso ao histórico de resultados obtidos. Outra parte importante do *master* é o agendador que ao receber

múltiplas requisições de testes, os enfileira por ordem de chegada e os libera a partir da disponibilidade das placas. A comunicação entre *master* e *worker* é realizada a partir de *daemons* presente em ambas as estruturas via HTTP (*Hypertext Transfer Protocol*). Por fim, o *worker* é responsável por estabelecer a conexão física com DUT (*device under test*) e partir dos comandos recebidos do *master*, realizar os testes nas placas (LAVA, 2020).

Apesar de o sistema permitir um único *master*, múltiplos *workers* podem atuar paralelamente e possuir múltiplas placas conectadas simultaneamente. Essa estrutura permite que equipamentos possam ser testados ao mesmo tempo sem interferência entre si, além de dividir a carga de processamento entre múltiplos *workers*.

Para o processo de inicialização do teste do produto embarcado, o LAVA necessita fazer recebimento do arquivo de configuração do teste em formato YAML (*YAML Ain't Markup Language*). A ferramenta recebe este arquivo de forma codificada em XML (*Extensible Markup Language*) via protocolo de chamada remota de procedimento (XML-RPC).

Ao receber o teste, o LAVA o armazena em uma fila de testes a serem executados. Em seguida, é verificado se o tipo do equipamento especificado no teste está disponível. Caso aprovado, é inicializada a etapa de *Deploy* no qual os arquivos necessários para o teste são baixados. Dependendo do sucesso da etapa de *Deploy*, se inicia a etapa de *Boot* ao qual uma série de métodos são fornecidos pela ferramenta. Nesta é realizada a gravação no equipamento com código a ser testado, além da ativação de possíveis equipamentos geradores de sinais externos. Por último, é realizado o monitoramento da placa onde novamente é disponibilizada uma série de métodos para diferentes tipos de análises dos resultados dos testes.

Apesar do LAVA fornecer todo ambiente necessário para inicialização dos testes e seu monitoramento, ainda é exigido especificações por parte do desenvolvedor quanto ao tipo do teste a ser executado e os resultados esperados. Sem esta configuração, a ferramenta de teste não possui meios a fim de validar ou reprovar os DUTs conectados, seguimento indispensável para o método da integração contínua.

2.4 Tipos de teste para embarcados

Testes realizados durante o desenvolvimento de um projeto embarcado é a principal finalidade a ser alcançada pela integração contínua, sendo assim, são primordiais para um desenvolvimento ágil, estável e maduro. Por esse motivo o entendimento dos tipos de testes e a importância dos mesmos se fazem de grande valor em ambiente de desenvolvimento. Os tipos de testes a serem abordados serão: testes unitários, testes de integração e testes de estresse.

2.4.1 Testes unitários

Como explicitado no próprio nome, testes unitários tem o objetivo de testar especificamente uma unidade isolada do projeto. Quando aplicada ao contexto de produtos embarcados, a unidade refere-se muitas vezes a camadas de baixo nível como teste de uma unidade lógica aritmética do processador ou até mesmo simples manipulações de dados em espaços de memória. Entretanto, conforme o contexto a qual é aplicado, a definição de uma unidade pode variar. Um exemplo seria um teste unitário de um sensor conectado a um processador. Neste exemplo, o teste de aquisição de valores provenientes do sensor pode ser definido como um teste unitário, contudo se analisado a partir de uma camada de mais baixo nível, poderia também ser considerado como um teste de acionamento do sensor juntamente ao teste de comunicação do mesmo com o processador, o que o classificaria como um teste de integração. Conseqüentemente, a definição de uma unidade do teste unitário pode variar de acordo com o projeto a ser desenvolvido.

2.4.2 Testes de integração

Ao contrário dos testes unitários, os testes de integração são executados utilizando diversas unidades simultaneamente. A finalidade do teste é verificar a interação entre diferentes módulos e garantir o funcionamento dos grupos testados como um todo, além de expor possíveis pontos de falhas onde o sistema pode apresentar erros ou instabilidades na comunicação entre tais unidades. Na grande maioria das vezes, os testes de integração devem ser realizados após a confirmação

de sucesso dos testes unitários, de forma a garantir que todos os módulos presentes neste teste coletivo foram individualmente validados.

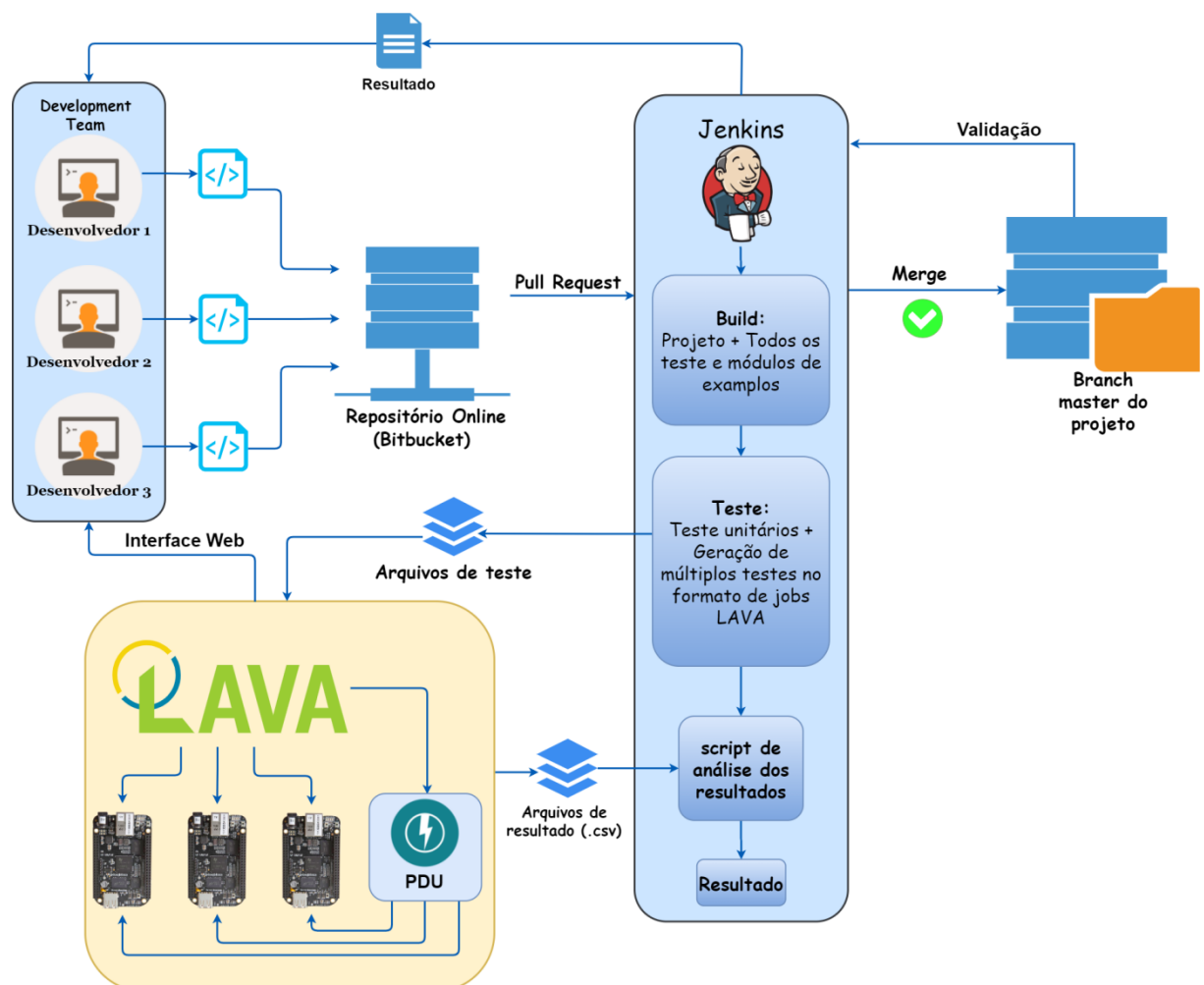
2.4.3 Testes de estresse

No contexto de desenvolvimento de produtos embarcados, testes de estresse são fundamentais para garantia de qualidade e longevidade dos equipamentos. Estes testes variam radicalmente conforme o tipo do produto desenvolvido e deve buscar impor os limites previstos durante as definições do projeto. Um teste comum de estresse de produtos embarcados é a execução prolongada das funcionalidades do equipamentos por diversos dias a fim de simular seu funcionamento em campo.

3 DESENVOLVIMENTO

Como ponto inicial para o desenvolvimento do sistema, foi iniciado a procura por material didático relevante ao tema de integração contínua e sistemas embarcados. O estudo foi realizado por meio de leitura de artigos científicos, livros, palestras e principalmente conferências *online*. Devido ao tema do projeto, grande parte do conteúdo encontrado tem origem nos fornecedores das ferramentas disponíveis no mercado, onde a documentação e informações podem ser facilmente encontradas nos seus respectivos sites. Com a ideia básica do funcionamento das ferramentas envolvidas no projeto, criou-se uma estrutura inicial do sistema de integração contínua a ser implementada representada na figura 3.

Figura 3 - Estrutura projetada da integração contínua



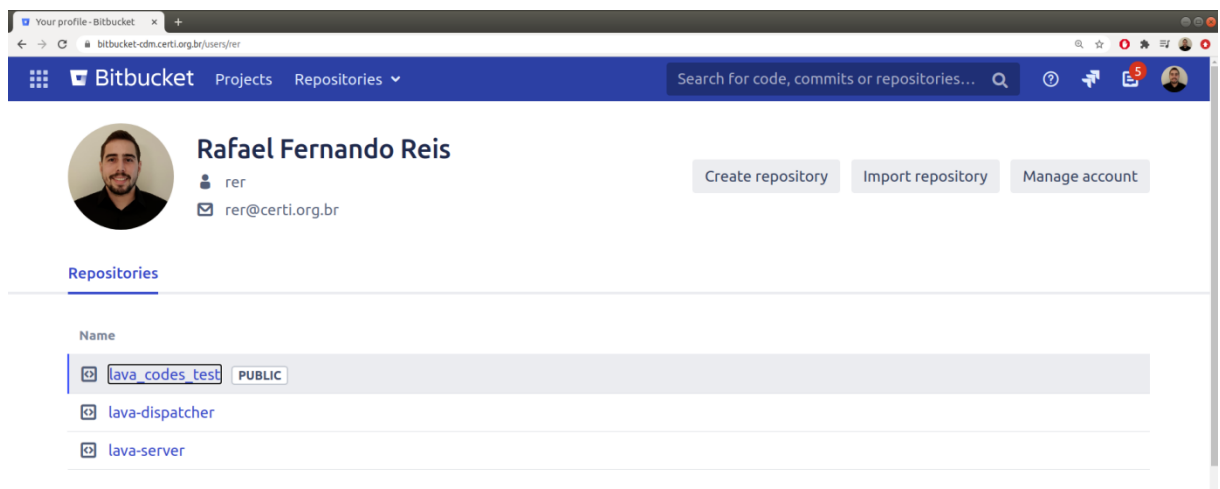
Fonte: Autor (2020)

Seguindo o fluxo apresentado na figura 3, a integração contínua é iniciada a partir da inclusão de novos arquivos ao repositório online. A partir desse momento, a ferramenta Jenkins é iniciada no qual realiza o *build* do projeto, gerando os arquivos executáveis necessários e inicializando em sequência os testes na ferramenta LAVA. Por fim a partir da avaliação dos resultados dos testes, o Jenkins valida ou não a inclusão dos novos arquivos ao repositório.

3.1 Repositório de arquivos online e Jenkins

A primeira ferramenta a ser escolhida para implementação do novo sistema automatizado foi o repositório de arquivos. Entre os muitos disponíveis para utilização em projetos embarcados, para o sistema proposto foi escolhido a ferramenta Bitbucket devido a sua interface amigável e facilidade de integração com a ferramenta de integração contínua. A figura 4 mostra a interface de usuário do repositório online.

Figura 4 - Repositório online Bitbucket



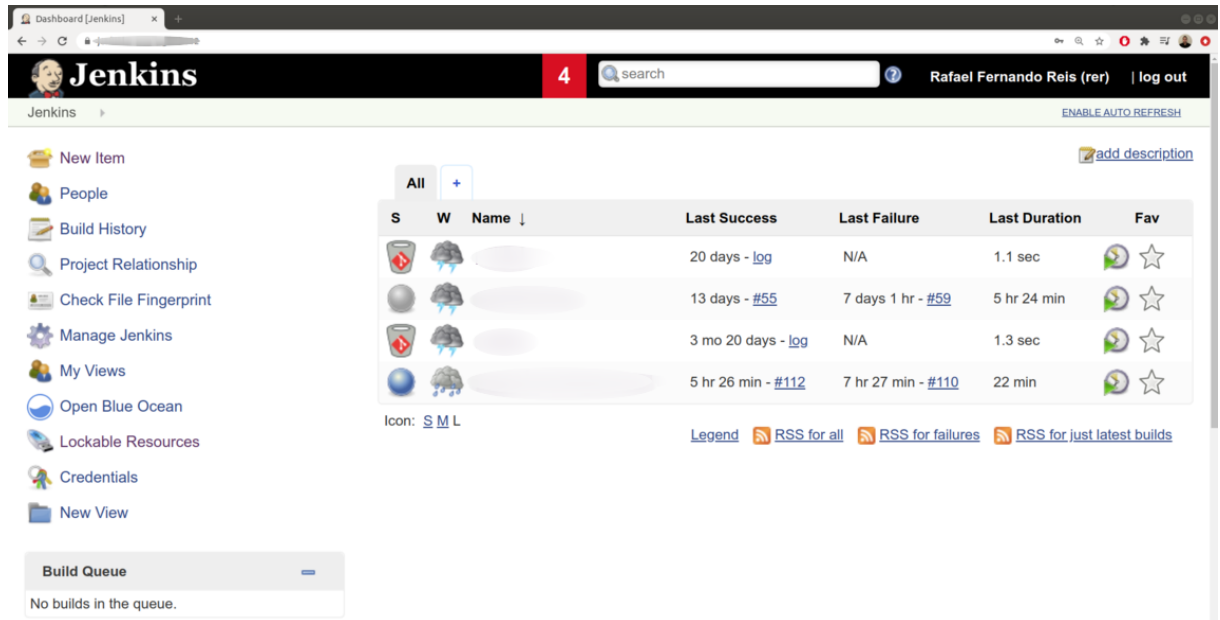
Fonte: Bitbucket Website (2020).

Com o repositório escolhido se torna mais fácil a organização das outras ferramentas já que elas também podem ser armazenadas de forma online similarmente aos futuros projetos.

Em seguida se iniciou a instalação e configuração do Jenkins. Sendo a principal ferramenta da integração contínua, a comunicação entre o Jenkins e o

repositório escolhido é fundamental para o funcionamento do sistema como um todo. A figura 5 mostra a interface da ferramenta utilizada.

Figura 5 - Interface Jenkins



Fonte: Jenkins Server (2020).

Com a ferramenta instalada é possível começar as configurações iniciais. O primeiro passo para a implementação da integração contínua no Jenkins se faz pela conexão do repositório à ferramenta. Existem muitas formas de realizar este processo sendo uma delas pela criação de um *job* que realiza o download de todo o repositório em sua execução. Para isso é necessário criar um *job* e na aba de configurações de *Pipeline* especificar a qual repositório a ferramenta deve estar conectada. A Figura 6 mostra a aba de configuração no qual se devem passar as credenciais do repositório e seu URL.

Figura 6 - Configuração de pipeline do Jenkins

Pipeline

Definition: Pipeline script from SCM

SCM: Git

Repositories:

- Repository URL: (Error: Please enter Git repository.)
- Credentials: - none - (Add button)
- Advanced... button
- Add Repository button

Branches to build:

- Branch Specifier (blank for 'any'): develop (Add button)
- Branch dropdown: (Auto)

Repository browser: (Auto)

Additional Behaviours: Add

Script Path: Jenkinsfile

Lightweight checkout:

Buttons: Save, Apply

[Pipeline Syntax](#)

Fonte: Jenkins Server (2020).

Outras configurações importantes são as definições do *branch* do repositório a ser utilizado e o *Script Path*, o qual irá definir o nome do arquivo que será executado após o *download* do repositório. Nesse arquivo, serão especificados os passos do processo de *build* e teste do projeto. Essas especificações deverão ser escritas utilizando uma linguagem proprietária do Jenkins. A figura 7 apresenta um exemplo simples da linguagem.

Figura 7 - Exemplos da linguagem declarativa de pipeline do Jenkins

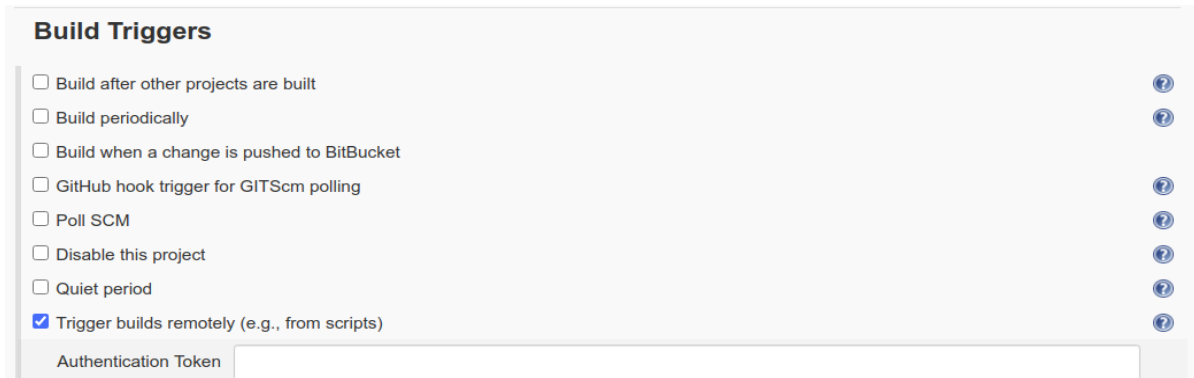
```
Jenkinsfile (Declarative Pipeline)
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        echo 'Building..'
      }
    }
    stage('Test') {
      steps {
        echo 'Testing..'
      }
    }
    stage('Deploy') {
      steps {
        echo 'Deploying....'
      }
    }
  }
}
```

Fonte: Jenkins Website (2020).

Para validação da união das duas ferramentas é interessante realizar o build do projeto a partir do Jenkins. Para tal tarefa, se faz necessário definir quando exatamente o *job* será realizado, ou seja, a partir de qual tipo de modificação do repositório deve-se iniciar a execução do *job*. Novamente a ferramenta utilizada apresenta diversas formas de realizar esta configuração que podem ser exploradas a partir da necessidade do projeto. A figura 8 apresenta a aba da configuração de possíveis *triggers* para o *job* do Jenkins.

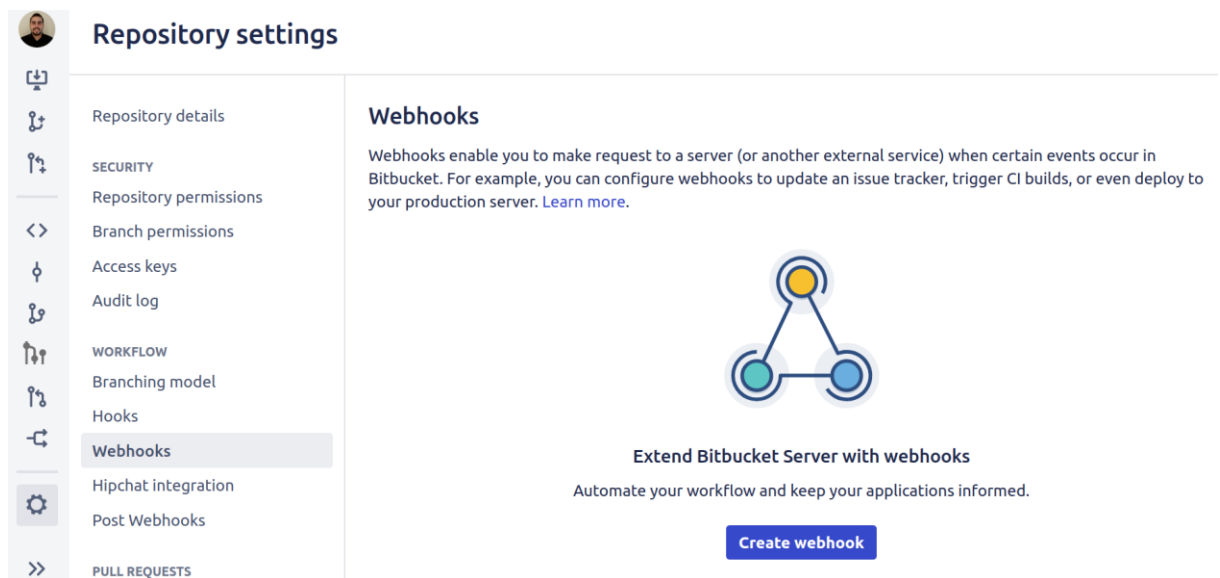
Figura 8 - Aba de configuração de trigger do Jenkins



Fonte: Jenkins Server (2020).

Uma das formas comumente utilizadas de *trigger* é a partir de requisições de *merges* no *branch* principal do repositório do projeto. Para esta configuração se faz necessário ativar as funcionalidades de *Webhooks* do bitbucket. Este modo de ativação utiliza um *token* de ativação e possibilita a execução de um *trigger* no job do Jenkins a partir de diferentes tipos de modificações no repositório. A figura 9 mostra onde se pode encontrar essa funcionalidade nas configurações do repositório.

Figura 9 - Aba de Webhooks do Bitbucket



Fonte: Bitbucket Website (2020).

Com a comunicação entre o repositório e o Jenkins estabelecido, a estrutura base da integração contínua se forma. A partir deste ponto, com ambas as ferramentas

corretamente configuradas, se faz possível realizar o *build* automaticamente de um projeto. Entretanto, para uma implementação robusta, a etapa de testes se faz necessária.

3.2 Automatização de testes

Para a etapa de teste, a ferramenta utilizada foi o LAVA. Começando pela instalação, um dos métodos mais simples para se iniciar a ferramenta é utilizando o Docker. Na documentação do LAVA é disponibilizado um Docker *container* que pode ser utilizado para facilitar a instalação da ferramenta e suas dependências. Como base para uma implementação inicial, foi-se utilizado o artigo “*Setting up a board farm with lava and docker*” de Dan Rue. A figura 10 mostra a interface do software utilizando as configurações do artigo citado.

Figura 10 - Interface web do LAVA

The screenshot shows the LAVA web interface. At the top, there is a navigation bar with links for Home, Results, Scheduler, API, and Help. The instance name is 'staging' and there is a 'Sign In' button. Below the navigation bar, the main content area starts with a 'Welcome to LAVA' section, followed by a paragraph describing LAVA as an automated validation architecture. Below this is a section titled 'About the LAVA staging instance' with a short description. The next section is 'LAVA components', which lists several key features: Results, Scheduler, API, Help, and Sign In. At the bottom, there are two columns of buttons: 'Guides to LAVA' and 'Test using LAVA'. The 'Guides to LAVA' column contains buttons for 'Introduction to LAVA', 'Administering a LAVA instance', and 'Developing LAVA'. The 'Test using LAVA' column contains buttons for 'Use cases and worked examples.', 'Writing a LAVA test definition.', and 'Logging into a LAVA device.'

Fonte: LAVA Server (2020).

Tendo o artigo como ponto de partida, iniciou-se as modificações nas configurações do software para atender a execução de testes simples em placas de desenvolvimento. Foi utilizada uma placa STM32H743ZI, modelo mostrado na figura 11, para realização dos testes iniciais.

Figura 11 - Placa de desenvolvimento STM32H743ZI



Fonte: Mouser Electronics Website (2020).

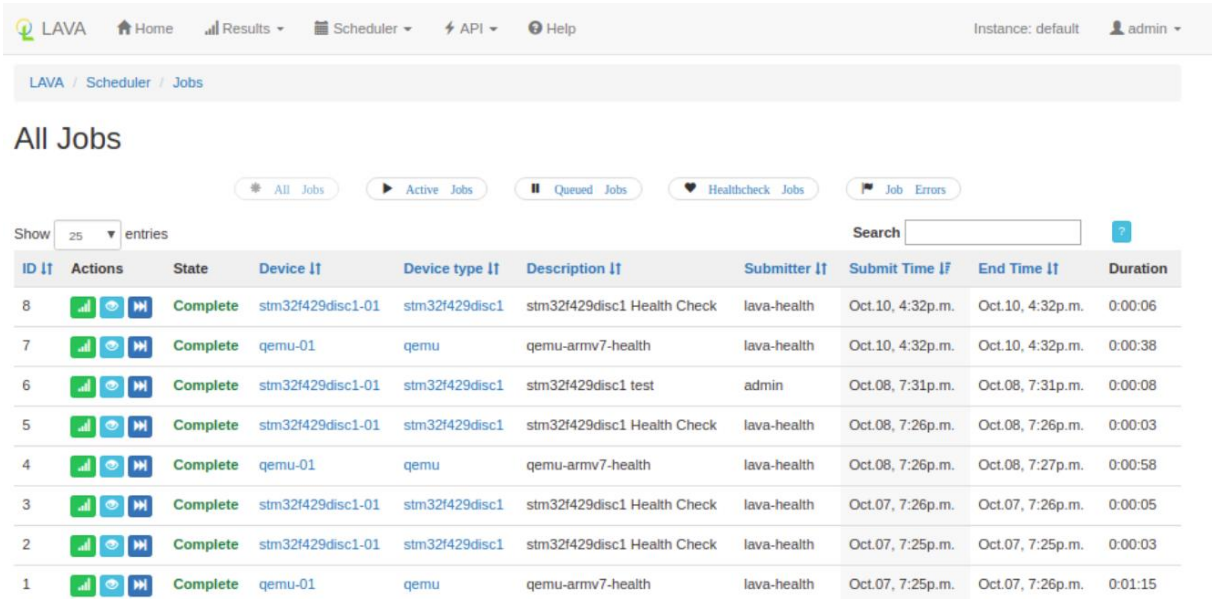
Para configuração desta placa no sistema, fez-se necessário a criação de três arquivos principais: *Device Type*, *Device* e o *Health Check*. Tais arquivos são utilizados para definir qual o tipo da placa que está conectada, qual é a placa a ser utilizada no caso de múltiplas placas iguais estarem conectadas e qual deve ser o teste executado para verificação do estado atual da placa. Tais arquivos de configuração são essenciais para cada placa conectada ao sistema e necessitam estar configurados corretamente para execução do software. Os anexos A, B e C foram os arquivos de *Device Type*, *Device* e o *Health Check* utilizados respectivamente.

Com a placa devidamente configurada para o sistema, foi possível criar um arquivo de teste LAVA a ser executado nesta placa. Entretanto, para que o sistema ficasse completo, o teste deve ser inicializado a partir da integração contínua. Para a integração das ferramentas, foram utilizados scripts Python que geram o arquivo de configuração de teste do LAVA e o enviam via protocolo XML-RPC para o LAVA *master*. Os scripts são chamados no arquivo Jenkinsfile no estágio de teste, resultando na automação do envio a partir de modificações no repositório. Os scripts que realizam

a criação do arquivo YAML e o envio via XML-RPC podem ser observados no apêndice A e B respectivamente.

O método escolhido para gravação do firmware foi a utilização do OpenOCD (Open On-Chip Debugger). Já para o monitoramento dos produtos embarcados foi definida a detecção via expressões regulares da serial do equipamento, no qual o resultado dos testes é dependente da identificação de mensagens de erro emitidas pela placa testada. Com o envio dos testes ao LAVA é possível observar sua execução a partir da interface web disponibilizada. A figura 12 mostra o resultado dos testes recebidos pelo LAVA e executado na placa STM32H743ZI. Os testes iniciais foram configurados de modo a realizar apenas a gravação da placa sem qualquer tipo de monitoramento da aplicação executada a fim de confirmar o funcionamento básico da ferramenta.

Figura 12 - Resultado dos testes na interface web



The screenshot shows the LAVA Scheduler interface. At the top, there is a navigation bar with 'LAVA', 'Home', 'Results', 'Scheduler', 'API', and 'Help'. The user is logged in as 'admin'. Below the navigation bar, the breadcrumb 'LAVA / Scheduler / Jobs' is visible. The main heading is 'All Jobs'. There are filters for 'All Jobs', 'Active Jobs', 'Queued Jobs', 'Healthcheck Jobs', and 'Job Errors'. A search bar is present. The table below lists jobs with columns for ID, Actions, State, Device ID, Device type, Description, Submitter, Submit Time, End Time, and Duration. All jobs shown are in a 'Complete' state.

ID	Actions	State	Device ID	Device type	Description	Submitter	Submit Time	End Time	Duration
8		Complete	stm32f429disc1-01	stm32f429disc1	stm32f429disc1 Health Check	lava-health	Oct.10, 4:32p.m.	Oct.10, 4:32p.m.	0:00:06
7		Complete	qemu-01	qemu	qemu-armv7-health	lava-health	Oct.10, 4:32p.m.	Oct.10, 4:32p.m.	0:00:38
6		Complete	stm32f429disc1-01	stm32f429disc1	stm32f429disc1 test	admin	Oct.08, 7:31p.m.	Oct.08, 7:31p.m.	0:00:08
5		Complete	stm32f429disc1-01	stm32f429disc1	stm32f429disc1 Health Check	lava-health	Oct.08, 7:26p.m.	Oct.08, 7:26p.m.	0:00:03
4		Complete	qemu-01	qemu	qemu-armv7-health	lava-health	Oct.08, 7:26p.m.	Oct.08, 7:27p.m.	0:00:58
3		Complete	stm32f429disc1-01	stm32f429disc1	stm32f429disc1 Health Check	lava-health	Oct.07, 7:26p.m.	Oct.07, 7:26p.m.	0:00:05
2		Complete	stm32f429disc1-01	stm32f429disc1	stm32f429disc1 Health Check	lava-health	Oct.07, 7:25p.m.	Oct.07, 7:25p.m.	0:00:03
1		Complete	qemu-01	qemu	qemu-armv7-health	lava-health	Oct.07, 7:25p.m.	Oct.07, 7:26p.m.	0:01:15

Fonte: LAVA Server (2020).

3.2.1 Unidade de controle de acionamento e teste

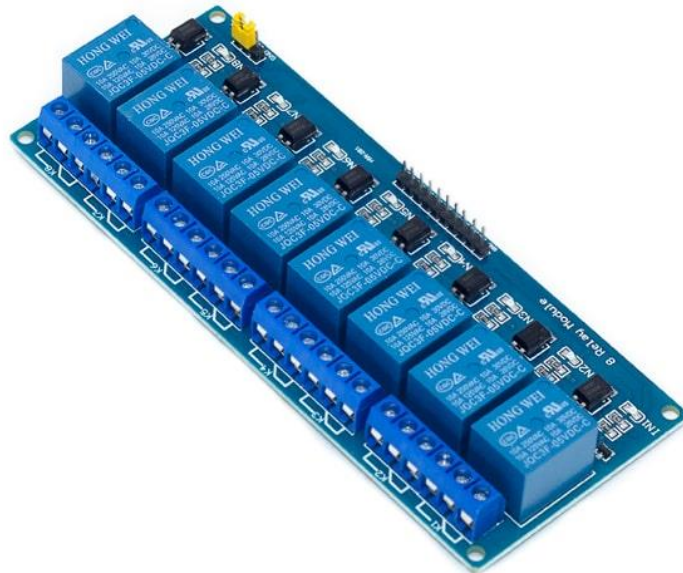
Neste ponto do sistema desenvolvido, observa-se que grande parte da integração contínua já está implementada. Entretanto, muitas vezes nos testes de sistemas embarcados, a manipulação de periféricos externos que interagem com o

equipamento a ser testado e o chamado *hard reset* da placa se fazem necessários. Para suprir tal necessidade de forma automatizada para o sistema, uma unidade secundária de controle de acionamento e teste se faz indispensável. Novamente, foi utilizada a placa STM32H743ZI para fazer este controle dos DUTs.

Para esta placa de controle foi escolhido o sistema operacional Mbed OS. A escolha deve-se principalmente a característica de desenvolvimento multiplataforma fornecida pelo SO, que permite a migração do código base entre diferentes microprocessadores contanto que a placa de desenvolvimento esteja na lista de placas permitidas pelo SO. Com esta definição, foi-se implementado uma CLI (Command Line Interface) que a partir de comandos específicos enviados por comunicação serial a placa, executa uma ação de controle definida. Tal implementação oferece ao usuário uma pluralidade de possíveis implementações futuras, onde dependendo do projeto e testes a serem executados, novos comandos podem ser implementados na placa. O apêndice C mostra o código desenvolvido da CLI para alguns comandos pré-definidos.

Em conjunto à CLI desenvolvida, a fim de possibilitar a alimentação de energia do DUT, função necessária para um *hard reset*, foi utilizada uma placa de relés. Este hardware permite o controle da alimentação de energia e a manipulação de periféricos externos a partir dos GPIOs da placa de controle, que com a utilização dos comandos da CLI, possibilita que este processo seja controlado a partir do sistema de integração contínua, especificamente pelo LAVA. A figura 13 mostra o modelo da placa de relés utilizada.

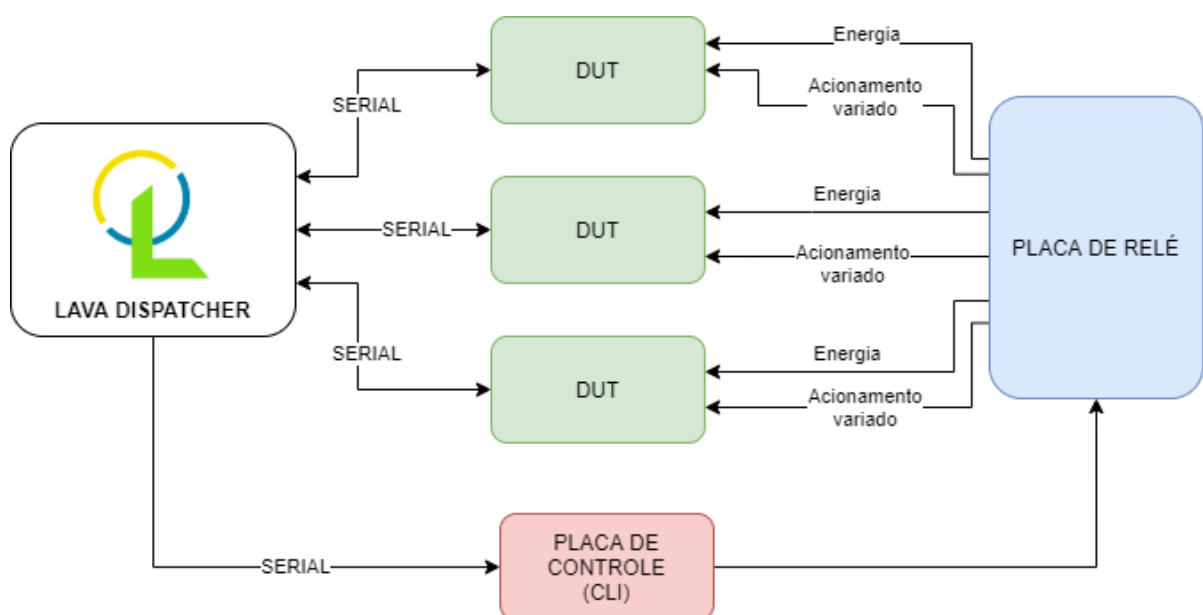
Figura 13 - Placa de relés



Fonte: Filipeflop Website (2020)

Com a adição da placa de relés, a estrutura de controle dos sistemas testados é finalizada e a figura 14 mostra o panorama final das conexões.

Figura 14 - Estrutura de controle dos DUTs



Fonte: Autor (2020).

4 APRESENTAÇÃO DOS RESULTADOS

Com todo o sistema completo, iniciou-se a aplicação do sistema de integração contínua em um projeto de desenvolvimento de software embarcado. Esta implementação foi realizada dentro da Fundação Centros de Referência em Tecnologias Inovadoras (CERTI) em Florianópolis no Centro de Convergência Digital e Mecatrônica (CDM). Este capítulo tem como objetivo relatar os resultados obtidos a partir da utilização do sistema desenvolvido dentro da empresa. Devido a acordo de confidencialidade, não será possível detalhar o produto desenvolvido, apenas os resultados referentes a utilização do sistema implantado.

Para integração do sistema no projeto, foi disponibilizado um armário destinado para conter os protótipos dos sistemas embarcados juntamente à placa de controle dos testes. A figura 15 mostra o armário com todo o sistema desenvolvido.

Figura 15 - Armário de teste

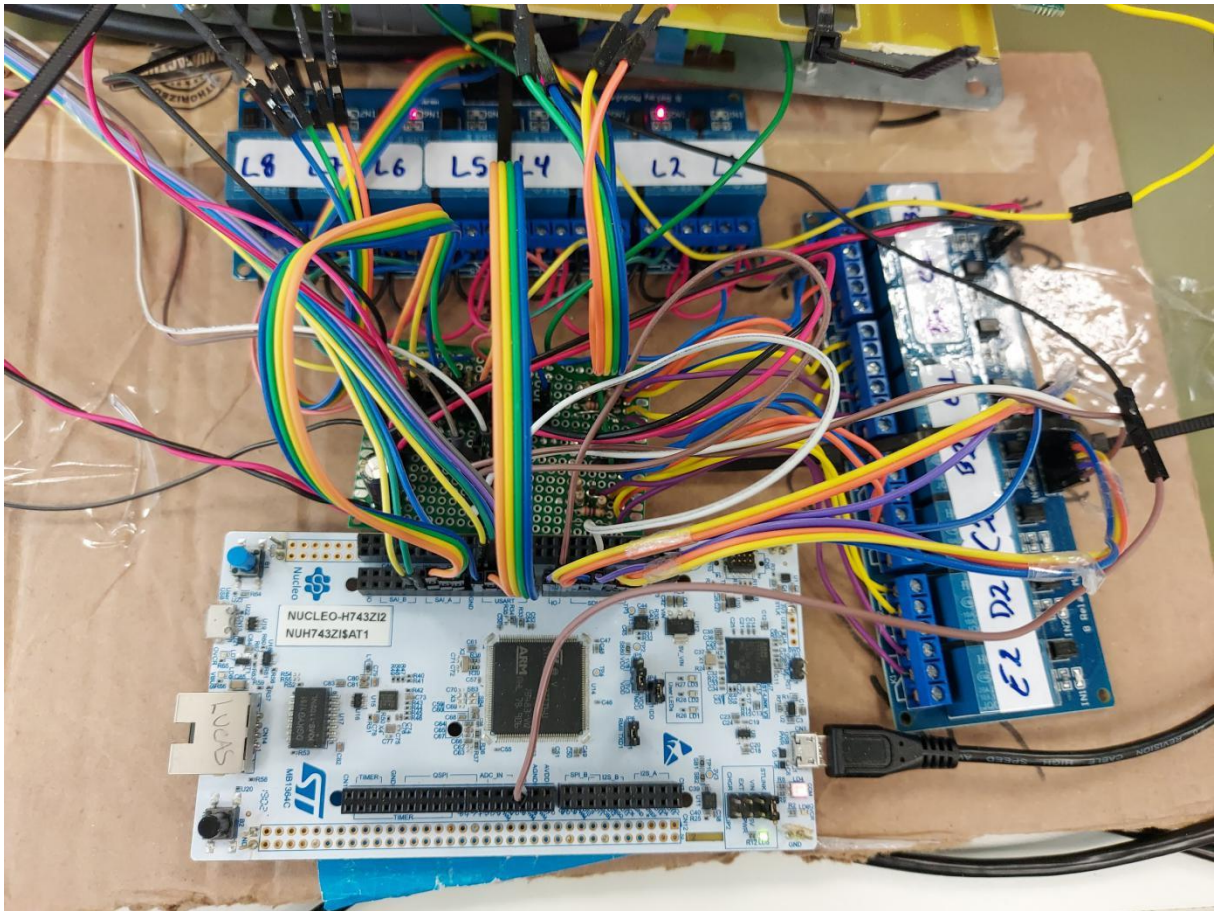


Fonte: Autor (2020).

4.1 Placa de controle desenvolvida

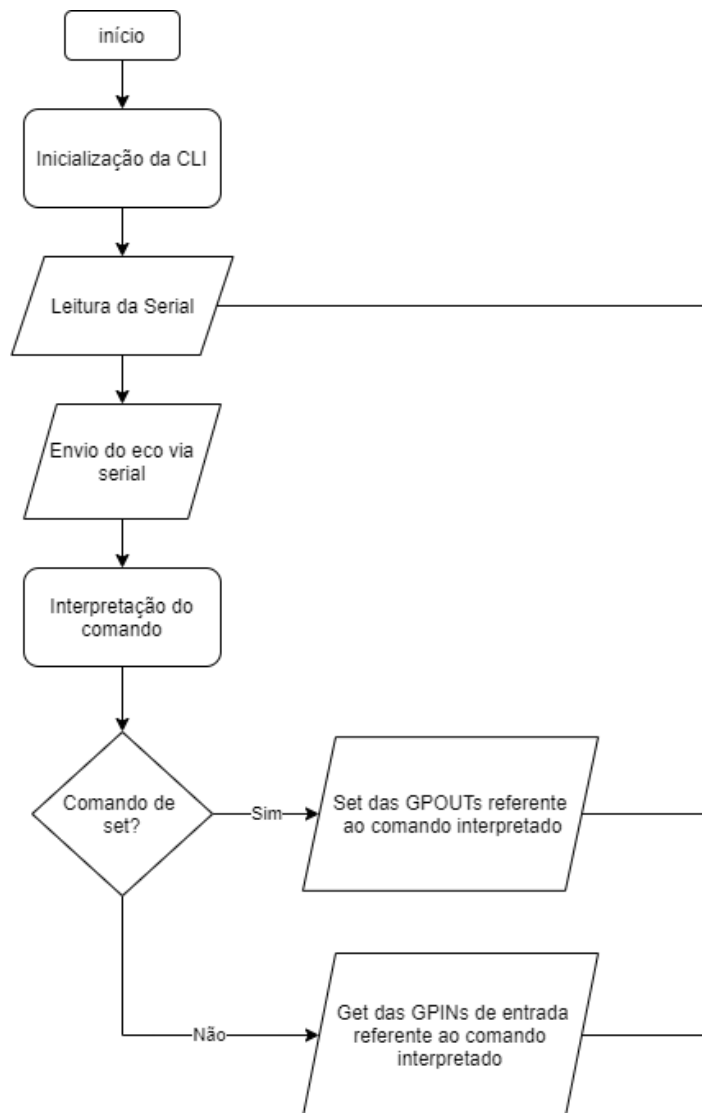
Para o sistema de controle de testes, a placa utilizada foi a STM32H743ZI como previsto. A figura 16 mostra a placa de controle utilizada para testar o produto embarcado em desenvolvimento e as placas de relés utilizadas.

Figura 16 - Placa de controle



Fonte: Autor (2020).

Com a placa de controle e a placa de relés conectadas, foram desenvolvidos comandos de controle de tensão em diferentes pontos do produto embarcado. Levando em consideração a velocidade de acionamento dos relés, em torno de 10 milissegundos, cada ponto da placa foi mapeado para um comando da CLI, possibilitando que um valor específico de tensão pré-configurado fosse aplicado em diferentes pontos. De forma semelhante, também foram desenvolvidos comandos na CLI para a confirmação do acionamento dos diodos emissores de luz do produto. A figura 17 mostra o diagrama simplificado do funcionamento da CLI.

Figura 17 - Fluxograma simplificado da CLI

Fonte: Autor (2020).

Como pode ser observada na figura 17, após a inicialização, a placa de controle realiza a leitura da serial e a envia de volta como um eco. Utilizando os dados lidos, a CLI realiza a interpretação do comando ao qual é diferenciado entre comandos de *set* ou *get*. A partir do comando interpretado, é realizado a escrita ou leitura da GPIO correspondente ao comando.

4.2 Erros detectados

Durante a utilização do sistema, foi observada a serventia do sistema como um todo, que como esperado, realizou a detecção de diversos tipos de erros durante a fase de desenvolvimento do projeto embarcado.

4.2.1 Erros no *build*

Sendo o erro mais recorrente durante a fase de desenvolvimento observada, erros no build são facilmente detectados no Jenkins. Como a causa do erro pode ser de origem extremamente variada, a fácil visualização dos motivos das falhas na plataforma da ferramenta auxiliaram na correção dos mesmos.

Um dos erros mais detectados foi a falta de atualização dos submódulos do repositório. Com a modificação de diversos arquivos de submódulos diferentes, se faz necessário que se atualize a *hash* dos novos *commits* no repositório principal do projeto. Esse erro foi detectado diversas vezes pela integração contínua que ao utilizar submódulos desatualizados, finalizou em erro e informou o desenvolvedor. Outro problema muito detectado foi a da implementação incorreta de periféricos que não consideravam a característica multiplataforma do projeto. Como o código do projeto foi separado em *targets*, casos ocorreram onde os desenvolvedores não testaram o build do sistema para todos os *targets* possíveis antes do envio das modificações ao repositório, erro que foi detectado dentro do estágio de build no Jenkins.

Observou-se também que como essas falhas eram detectadas antes da inclusão no código fonte do projeto, a garantia de sucesso de build no código fonte foi validada e comprovada durante toda a etapa de desenvolvimento do software.

4.2.2 Erros nos testes unitários

Durante o desenvolvimento do projeto embarcado, diversos testes unitários foram desenvolvidos e executados na integração contínua projetada. Na execução de tais testes observou-se que após a primeira validação dos componentes testados,

difícilmente presenciaram-se mais erros. Como estes testes isolam os componentes na sua execução, novas implementações no software base raramente resultaram na falha de testes prévios. Entretanto, devido mudanças repentinas nas definições do produto durante a fase de desenvolvimento, o refatoramento do código se fez necessário. Refatorar o código pode modificar a estrutura de classes e métodos já validados, sendo assim, os testes unitários realizados previamente à refatoração podem falhar. Este erro foi detectado a partir da integração contínua, no qual se fez necessário a modificação do teste a fim de atender a nova estruturação.

Para modificações de hardware esses testes se tornaram fundamentais. Como o projeto embarcado desenvolvido também contava com um hardware dedicado, modificações de hardware em componente já implementadas em software foram a origem de diversos erros nos testes unitários detectados inicialmente pelo sistema de integração contínua. Muitas das vezes que essas modificações eram necessárias, já era esperado que os testes unitários relacionados fossem falhar, todavia, a confirmação por meio da integração contínua agilizou o processo de correção por informar rapidamente os desenvolvedores que não estavam cientes das modificações de hardware. Um dos momentos onde este erro foi observado foi na modificação do expensor de GPIO do produto. Inicialmente a comunicação com o expensor era feita via I2C (Inter-Integrated Circuit), mas devido mudanças no hardware do produto, foi modificada para SPI (Single Peripheral Interface). Tal mudança foi acusada no sistema de integração contínua no teste unitário de comunicação com o expensor e modificada posteriormente.

4.2.3 Erros de gravação de binário

Outro erro que foi detectado durante a integração contínua foi o erro no processo de gravação do binário do projeto no hardware dedicado. Como nem todos os desenvolvedores de um projeto embarcado necessitam ter contato direto com o hardware, o processo de gravação do novo software na placa é muitas vezes executado apenas pela integração contínua. Com o aumento das funcionalidades e implementações de software, o tamanho pode ultrapassar a capacidade interna de memória do processador, erro que foi observado durante a etapa de desenvolvimento devido a inclusão de bibliotecas desnecessárias no software do equipamento. Esse

problema foi facilmente detectado no LAVA que relatou e informou o erro no momento de falha de gravação, agilizando sua correção antes da inclusão no repositório base do projeto.

4.2.4 Erros nos testes de estresse

Os testes de estresse realizados no projeto consistiram no uso contínuo do equipamento por até 100 horas consecutivas. Nesses testes, diversos agentes externos foram controlados com a utilização da placa de controle a fim de simular a utilização de um usuário. Durante a maior parte da etapa de desenvolvimento, estes testes eram iniciados e finalizados aos fins de semana a fim de liberar as placas conectadas ao sistema para outros tipos de testes.

Os principais erros encontrados foram problemas mecânicos devido às vibrações do equipamento e também erros devido à alocação de memória do processador. Como muitas vezes o problema de alocação é um erro acumulativo, o erro resultante pode ocorrer horas após a inicialização do equipamento, tornando o teste de estresse ainda mais relevante.

De forma geral, a automatização do teste de estresse foi um dos fatores que trouxeram mais agilidade ao desenvolvimento do projeto. Como esse teste geralmente é um processo longo com muitas horas de execução, a automatização deste processo minimizou quase por completo a necessidade de um desenvolvedor testando o equipamento manualmente por diversas horas.

5 CONCLUSÃO

Com o estudo da metodologia de integração contínua, implementação do sistema desenvolvido e avaliação dos resultados e erros encontrados durante a fase de desenvolvimento de um sistema embarcado, pode-se afirmar que os objetivos específicos do trabalho foram alcançados.

De forma geral, o trabalho apresentado amplia a compreensão da metodologia ágil de integração contínua e suas vantagens na utilização no desenvolvimento de sistemas embarcados. A partir dos resultados obtidos, foi possível avaliar as soluções propostas pela metodologia para as problemáticas do desenvolvimento de sistema embarcados levantadas previamente. Com a utilização do sistema proposto foi possível notar uma tendência de um fluxo de desenvolvimento mais ágil, organizado e robusto principalmente pelo fato do código fonte do projeto estar funcional a todo o momento. Devido principalmente à automação dos testes, também foi percebido que a quantidade de erros introduzidos no código fonte foram reduzidos. Além disso, devida a utilização do hardware do sistema embarcado em desenvolvimento nos testes, problemas no projeto da placa foram encontrados e corrigidos rapidamente, validando o hardware e software simultaneamente durante toda a fase de desenvolvimento do projeto. Sendo assim, em concordância com a bibliografia utilizada no decorrer dos estudos, foi observado a eficácia da integração contínua no desenvolvimento de sistemas embarcados.

5.1 Sugestões de trabalhos futuros

Com o desenvolvimento do sistema proposto, sugestões de novas implementações adicionais foram realizadas a fim de aperfeiçoar ainda mais o sistema desenvolvido. Entre elas destacam-se:

- a) Desenvolver uma placa de controle de energia dedicada ao invés da utilização de placas de desenvolvimento como a STM32H743ZI.
- b) Adicionar medições de tensão e corrente à alimentação da placa em desenvolvimento para levantamento de consumo energético durante os testes.

REFERÊNCIAS

DOCKER. WHAT is a container. Disponível em <https://www.docker.com/resources/what-container>. Acesso em 15 out. 2020.

DUVAL, Paul M.; MATYAS, Steve; GLOVER, Andrew. Continuous Integration: Improving Software Quality And Reducing Risk. [S. l.]: Pearson Education, 2007.

FOWLER, Martin. Continuous Integration. 2006. Disponível em <http://martinfowler.com/articles/continuousIntegration.html>. Acesso em 01 outubro 2020.

GOOGLE. Continuous Integration (CI). Disponível em: <https://cloud.google.com/solutions/continuous-integration>. Acesso em: 15 out. 2020.

JENKINS. Jenkins User documentation. Disponível em: <https://www.jenkins.io/doc/>. Acesso em 01 de outubro de 2020.

LAVA. Introduction to LAVA. Disponível em: <https://docs.lavasoftware.org/lava/index.html>. Acesso em 15 out. 2020.

MARWEDEL, Peter. Embedded System Design, 2. ed. Dortmund: Springer Publishing, 2011.

RUE, Dan. Setting up a board farm with LAVA and Docker. 2019. Disponível em: <https://therub.org/2019/03/05/setting-up-a-board-farm-with-lava-and-docker>. Acesso em: 15 out. 2020.

SMART, John Ferguson. Jenkins The Definitive Guide. Sebastopol: O'Really Media, 2011.

WADHWANI, Preeti; YADAV, Shubhangi. Embedded Systems Market Size By Component. Disponível em: <https://www.gminsights.com/industry-analysis/embedded-system-market>. Acesso em 01 de novembro de 2020.

APÊNDICES

APÊNDICE A – Script de criação de arquivo de teste LAVA

```
#!/usr/bin/env python3

import yaml, sys
from collections import OrderedDict

def setup_yaml():
    represent_dict_order = lambda self, data:
self.represent_mapping('tag:yaml.org,2002:map', data.items())
    yaml.add_representer(OrderedDict, represent_dict_order)

def main(argv):
    project_name = argv[1]
    device_type = argv[2]
    module = argv[3]
    exec_file = argv[4]

    data = OrderedDict({
        'device_type': device_type,
        'job_name': (project_name + " " + module + " LAVA Test"),
        'timeouts': {
            'job': {
                'minutes': 6},
            'action': {
                'minutes': 5},
            'connection': {
                'minutes': 2}
        },
        'priority': 'medium',
        'visibility': 'public',
        'actions': [
            {'deploy':
                {'to': 'tmpfs',
                 'images':
```

```

        {'binary':
            {'url':
                '<lava_ip_port>' + project_name + "/" + device_type + "/" + module
+ "/" + exec_file}},
            'timeout': {'minutes': 3}},
        {'boot': {
            'method': 'openocd',
            'timeout': {
                'minutes': 2}},
        {'test': {
            'timeout': {
                'minutes': 2},
            'monitors':[OrderedDict({
                'name': 'Monitor-Test',
                'start': 'START TESTS',
                'end': 'END OF TESTS',
                'pattern':
'\[s+RUN\s+\]\s+(?P<test_case_id>(\w*\.\w*)|(\w*\Vw*\.\w*\Vw*)).*\n\[s+(?P<result>(OK
|FAILED))',
                'fixupdict': {
                    'OK': 'pass',
                    'FAILED': 'fail'},
            })
        ]
    }}
]
}
)
setup_yaml()
    with open('.build-scripts/scripts/lava/' + project_name + "_" + device_type + "_" +
module + ".yaml", 'w') as outfile:
        yaml.dump(data, outfile, default_flow_style=False)
        print("LAVA Job created successfully")

if __name__ == "__main__":
    main(sys.argv)

```

APÊNDICE B – Script de envio de arquivo de teste LAVA

```
#!/usr/bin/env python3

import os
import xmlrpc.client, yaml, simplejson, glob, csv, sys

#----- LAVA authentication-----
username = <user>
token = <token>
hostname = <hostname>
#-----

print('lava_script: Starting lava_script\n')
print('lava_script: LAVA Authentication:')
print('lava_script: Username:' + username)
print('lava_script: Token: '+ token )
print('lava_script: Hostname: '+ hostname +'\n')

server = xmlrpc.client.ServerProxy("http://%s:%s@%s/RPC2" % (username, token,
hostname), allow_none=True)
lava_version = server.system.version()
print('lava_script: Authentication PASSED\n')
print('lava_script: Sending LAVA Jobs:')

count_files=0
index=0
failed_flag=False
files=glob.glob(".build-scripts/scripts/lava/*.yaml")
jobid = [0]*len(files)

print('lava_script: Number of test files (.yaml) found: '+ str(len(files)))

for i in files:
    print('lava_script: '+ i +'\n')
```

```

f = open (i)
job = yaml.load(f, Loader=yaml.SafeLoader)
f.close()
os.remove(i)
config = simplejson.dumps(job)
jobid[count_files]=server.scheduler.submit_job(config)
count_files +=1
if (count_files == 0):
    print('lava_script: No tests found.')

else:
    print("lava_script: Jobs sent to LAVA server.")
    print("lava_script: Waiting for jobs to finish. This may take a while.")
    while (index < count_files) :
        job_state = str(server.scheduler.job_state(jobid[index]))

        while ("Finished" not in job_state):
            job_state = str(server.scheduler.job_state(jobid[index]))

        results = str(server.results.get_testjob_results_csv(jobid[index]))

        if "fail" in results:
            print('lava_script: Job ID: ' + str(jobid[index]) + ' State: Finished ' + 'Status:
FAILED\n')
            failed_flag = True
        else:
            print('lava_script: Job ID: ' + str(jobid[index]) + ' State: Finished ' + 'Status:
PASSED\n')
            index +=1

    if (failed_flag == True):
        print("lava_script: FAILED.")
        sys.exit(1)
    print("lava_script: SUCCESS.")
print('lava_script: End of lava_script')

```

APÊNDICE C – CLI desenvolvida

```
#include "mbed.h"
#include <map>
#include <string>

#define BUFFER_SIZE 32
#define OPEN_RELAY true
#define CLOSE_RELAY false

Thread thread;
static BufferedSerial serial_port(USBTX, USBRX);

static DigitalOut relay_B1(PC_8);
static DigitalOut relay_C1(PC_9);
static DigitalOut relay_D1(PC_10);
static DigitalOut relay_E1(PC_11);

static DigitalOut relay_B2(PC_12);
static DigitalOut relay_C2(PD_2);
static DigitalOut relay_D2(PG_2);
static DigitalOut relay_E2(PG_3);

static DigitalOut relay_SW1(PA_3);
static DigitalOut relay_SW2(PC_0);
static DigitalOut relay_SW3(PC_3);

static DigitalOut relay_CUR_LEAK(PG_1);

static DigitalIn relay_BLUE_1(PD_4);
static DigitalIn relay_YELLOW_1(PD_5);
static DigitalIn relay_GREEN_1(PD_6);
static DigitalIn relay_RED_1(PD_7);

static DigitalIn relay_BLUE_2(PE_5);
```

```

static DigitalIn relay_YELLOW_2(PE_4);
static DigitalIn relay_GREEN_2(PD_3);
static DigitalIn relay_RED_2(PE_2);

static DigitalOut relay_RESET(PB_1);
static DigitalOut relay_POWER(PE_3);

constexpr char invalid_response[] = "Invalid Command\n";

enum Commands {
    ON, OFF, SOFT_RESET, A12V, A9V, A6V, A3V, A0V, B12V, B9V, B6V, B3V, B0V,
    DIP000, DIP001, DIP010, DIP011, DIP100, DIP101, DIP110, DIP111,
    LEAK_CURRENT, BLUE_1, BLUE_2, YELLOW_1, YELLOW_2, GREEN_1,
    GREEN_2, RED_1, RED_2, EMPTY, INVALID};

void ReadCommand(char (*buffer)[BUFFER_SIZE]);
void ProcessCommand(char *);
void InitialState();
void CLI_Thread();

int main() {
    serial_port.set_baud(115200);
    serial_port.set_format(8, BufferedSerial::None, 1);

    InitialState();

    char msg[] = "\n\rInitializing CLI\n\r";
    serial_port.write(msg, sizeof(msg));
    thread.start(CLI_Thread);
}
/***** Functions *****/
void CLI_Thread() {
    char command[BUFFER_SIZE] = {0};
    while (true) {
        ReadCommand(&command);
    }
}

```

```

    ProcessCommand(command);
    memset(command, 0, sizeof(command));
}
}

```

```

Commands resolveCommand(std::string input) {
    static const std::map<std::string, Commands> commandString{
        {"set on", ON}, {"set off", OFF}, {"reset", SOFT_RESET}, {"set a 12", A12V},
        {"set a 9", A9V}, {"set a 6", A6V}, {"set a 3", A3V}, {"set a 0", A0V},
        {"set b 12", B12V}, {"set b 9", B9V}, {"set b 6", B6V}, {"set b 3", B3V},
        {"set b 0", B0V}, {"set dip 000", DIP000}, {"set dip 001", DIP001},
        {"set dip 010", DIP010}, {"set dip 011", DIP011}, {"set dip 100", DIP100},
        {"set dip 101", DIP101}, {"set dip 110", DIP110}, {"set dip 111", DIP111},
        {"leak current", LEAK_CURRENT}, {"get a blue", BLUE_1},
        {"get b blue", BLUE_2}, {"get a yellow", YELLOW_1},
        {"get b yellow", YELLOW_2}, {"get a green", GREEN_1},
        {"get b green", GREEN_2}, {"get a red", RED_1}, {"get b red", RED_2},
        {"\0", EMPTY},
    };
    auto itr = commandString.find(input);
    if (itr != commandString.end()) {
        return itr->second;
    }
    return INVALID;
}

```

```

void ReadCommand(char (*buffer)[BUFFER_SIZE]) {
    char *cursor = *buffer;
    char c[BUFFER_SIZE] = {0};

    while (true) {
        if (serial_port.readable()) {
            serial_port.read(c, sizeof(c));
            if (strlen(c) > 1) {
                *cursor = *c;
            }
        }
    }
}

```

```

    break;
}
if (*c == '\n' || *c == '\r') {
    *cursor++ = '\0';
    serial_port.write("\n", sizeof(char));
    serial_port.write("\r", sizeof(char));
    break;
}
serial_port.write(c, sizeof(char));
if (*c == 8 || *c == 127) { // backspace or del check
    serial_port.write("\b", sizeof(char));
    serial_port.write(" ", sizeof(char));
    serial_port.write("\b", sizeof(char));
    if (cursor >= *buffer + 1) {
        *cursor-- = '\0';
    }
} else {
    *cursor++ = *c;
}
}
}
}
}

void ProcessCommand(char *command) {
    switch (resolveCommand(command)) {
    case ON:
        relay_POWER = CLOSE_RELAY;
        break;
    case OFF:
        relay_POWER = OPEN_RELAY;
        break;
    case SOFT_RESET:
        relay_RESET = CLOSE_RELAY;
        wait_us(1000000); // 1 sec
        relay_RESET = OPEN_RELAY;
        break;

```


case A12V:

```
relay_B1 = OPEN_RELAY;  
relay_C1 = OPEN_RELAY;  
relay_D1 = OPEN_RELAY;  
relay_E1 = OPEN_RELAY;  
break;
```

case A9V:

```
relay_B1 = CLOSE_RELAY;  
relay_C1 = OPEN_RELAY;  
relay_D1 = OPEN_RELAY;  
relay_E1 = OPEN_RELAY;  
break;
```

case A6V:

```
relay_B1 = CLOSE_RELAY;  
relay_C1 = CLOSE_RELAY;  
relay_D1 = OPEN_RELAY;  
relay_E1 = OPEN_RELAY;  
break;
```

case A3V:

```
relay_B1 = CLOSE_RELAY;  
relay_C1 = OPEN_RELAY;  
relay_D1 = CLOSE_RELAY;  
relay_E1 = OPEN_RELAY;  
break;
```

case A0V:

```
relay_B1 = CLOSE_RELAY;  
relay_C1 = OPEN_RELAY;  
relay_D1 = OPEN_RELAY;  
relay_E1 = CLOSE_RELAY;  
break;
```

case B12V:

```
relay_B2 = OPEN_RELAY;  
relay_C2 = OPEN_RELAY;  
relay_D2 = OPEN_RELAY;  
relay_E2 = OPEN_RELAY;
```

```
break;
case B9V:
    relay_B2 = CLOSE_RELAY;
    relay_C2 = OPEN_RELAY;
    relay_D2 = OPEN_RELAY;
    relay_E2 = OPEN_RELAY;
    break;
case B6V:
    relay_B2 = CLOSE_RELAY;
    relay_C2 = CLOSE_RELAY;
    relay_D2 = OPEN_RELAY;
    relay_E2 = OPEN_RELAY;
    break;
case B3V:
    relay_B2 = CLOSE_RELAY;
    relay_C2 = OPEN_RELAY;
    relay_D2 = CLOSE_RELAY;
    relay_E2 = OPEN_RELAY;
    break;
case B0V:
    relay_B2 = CLOSE_RELAY;
    relay_C2 = OPEN_RELAY;
    relay_D2 = OPEN_RELAY;
    relay_E2 = CLOSE_RELAY;
    break;
case DIP000:
    relay_SW1 = CLOSE_RELAY;
    relay_SW2 = CLOSE_RELAY;
    relay_SW3 = CLOSE_RELAY;
    break;
case DIP001:
    relay_SW1 = OPEN_RELAY;
    relay_SW2 = CLOSE_RELAY;
    relay_SW3 = CLOSE_RELAY;
    break;
```

```
case DIP010:
    relay_SW1 = CLOSE_RELAY;
    relay_SW2 = OPEN_RELAY;
    relay_SW3 = CLOSE_RELAY;
    break;
case DIP011:
    relay_SW1 = OPEN_RELAY;
    relay_SW2 = OPEN_RELAY;
    relay_SW3 = CLOSE_RELAY;
    break;
case DIP100:
    relay_SW1 = CLOSE_RELAY;
    relay_SW2 = CLOSE_RELAY;
    relay_SW3 = OPEN_RELAY;
    break;
case DIP101:
    relay_SW1 = OPEN_RELAY;
    relay_SW2 = CLOSE_RELAY;
    relay_SW3 = OPEN_RELAY;
    break;
case DIP110:
    relay_SW1 = CLOSE_RELAY;
    relay_SW2 = OPEN_RELAY;
    relay_SW3 = OPEN_RELAY;
    break;
case DIP111:
    relay_SW1 = OPEN_RELAY;
    relay_SW2 = OPEN_RELAY;
    relay_SW3 = OPEN_RELAY;
    break;
case LEAK_CURRENT:
    relay_CUR_LEAK = CLOSE_RELAY;
    wait_us(1000000); // 1 sec
    relay_CUR_LEAK = OPEN_RELAY;
    break;
```

```
case BLUE_1:
    if (!relay_BLUE_1.read()) {
        serial_port.write("1", sizeof(char));
    } else {
        serial_port.write("0", sizeof(char));
    }
    break;
case BLUE_2:
    if (!relay_BLUE_2.read()) {
        serial_port.write("1", sizeof(char));
    } else {
        serial_port.write("0", sizeof(char));
    }
    break;
case YELLOW_1:
    if (!relay_YELLOW_1.read()) {
        serial_port.write("1", sizeof(char));
    } else {
        serial_port.write("0", sizeof(char));
    }
    break;
case YELLOW_2:
    if (!relay_YELLOW_2.read()) {
        serial_port.write("1", sizeof(char));
    } else {
        serial_port.write("0", sizeof(char));
    }
    break;
case GREEN_1:
    if (!relay_GREEN_1.read()) {
        serial_port.write("1", sizeof(char));
    } else {
        serial_port.write("0", sizeof(char));
    }
    break;
```

```
case GREEN_2:
    if (!relay_GREEN_2.read()) {
        serial_port.write("1", sizeof(char));
    } else {
        serial_port.write("0", sizeof(char));
    }
    break;
case RED_1:
    if (!relay_RED_1.read()) {
        serial_port.write("1", sizeof(char));
    } else {
        serial_port.write("0", sizeof(char));
    }
    break;
case RED_2:
    if (!relay_RED_2.read()) {
        serial_port.write("1", sizeof(char));
    } else {
        serial_port.write("0", sizeof(char));
    }
    break;
case EMPTY:
    break;
case INVALID:
    serial_port.write(invalid_response, sizeof(invalid_response));
    break;
}
}
void InitialState(void) {
    ProcessCommand((char *)"set on");
    ProcessCommand((char *)"reset");
    ProcessCommand((char *)"set a 12");
    ProcessCommand((char *)"set b 12");
    relay_CUR_LEAK = OPEN_RELAY;
}
```

ANEXO

ANEXO A – Arquivo DeviceType da placa STM32H743ZI

```

{# device_type: nucleo-h743zi2#}
{% extends 'base.jinja2' %}
{% block body %}

board_id: '{{ board_id|default('00000000') }}'

usb_vendor_id: '0483'
usb_product_id: '374b'
usb_sleep: {{ usb_sleep|default(10) }}

actions:
  deploy:
    methods:
      image:
        parameters:

boot:
  connections:
    serial:
  methods:
    openocd:
      parameters:
        command:
          # Can set 'openocd_bin_override' in device dictionary to
          # override location of OpenOCD executable to point to TI OpenOCD
          # if necessary
          {{ openocd_bin_override|default('openocd') }}
          #- 'openocd'
      options:
        file:
          - board/stm32h7x3i_eval.cfg

```

```

# Set 'openocd_scripts_dir_override' in device dictionary to
# point to TI OpenOCD scripts if necessary
search: [{{ openocd_scripts_dir_override }}]
command:
  - hla_serial 066DFF484851897767011422
  - init
  - targets
  - 'reset init'
  - 'flash write_image erase {BINARY}'
  - 'reset halt'
  - 'verify_image {BINARY}'
  - 'reset run'
  - shutdown
debug: 2
gdb:
  parameters:
    command: gdb-multiarch
    wait_before_continue: {{ wait_before_continue|default(5) }}
  openocd:
    arguments:
      - "{ZEPHYR}"
    commands:
      - target remote | openocd -c "gdb_port pipe" -f {OPENOCD_SCRIPT}
      - monitor reset halt
      - load
      - set remotetimeout 10000
    docker:
      use: {{ docker_use|default(False) }}
      container: '{{ docker_container|default('ti-openocd') }}'
      devices: {{ docker_devices|default([]) }}
{% endblock body %}

```


ANEXO B – Arquivo Device da placa STM32H743ZI

```
{% extends 'nucleo-h743zi2.jinja2' %}  
{% set connection_commands = {'uart0': 'telnet ser2net 5003'} %}  
{% set connection_list = ['uart0'] %}  
{% set connection_tags = {'uart0': ['primary', 'telnet']} %}  
  
{% set board_id = '000000001' %}
```

ANEXO C – Arquivo Health Check da placa STM32H743ZI

device_type: nucleo-h743zi2
job_name: nucleo-h743zi2 Health Check

timeouts:

job:

minutes: 6

action:

minutes: 5

connection:

minutes: 2

priority: medium

visibility: public

ACTION_BLOCK

actions:

- deploy:

timeout:

minutes: 5

to: tmpfs

images:

binary:

url: <image url>

- boot:

method: openocd

timeout:

minutes: 2