



UNIVERSIDADE FEDERAL DE SANTA CATARINA  
CAMPUS FLORIANÓPOLIS  
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Lucas Zacchi de Medeiros

**Análise e implementação de fluxo de automação de testes  
dos sistemas embarcados de Satélites Artificiais**

Florianópolis  
2022

Lucas Zacchi de Medeiros

**Análise e implementação de fluxo de automação de testes  
dos sistemas embarcados de Satélites Artificiais**

Monografia submetida ao Curso de Graduação em  
Ciências da Computação da Universidade Federal  
de Santa Catarina para a obtenção do título de ba-  
charel em Ciências da Computação.

Orientador: Prof. Dr. Eduardo Augusto Bezerra

Coorientador: Prof. Dr. Rafael de Santiago

Florianópolis

2022

À Maria, com amor. Minha companheira, amiga e por  
vezes revisora voluntária. Sua ajuda e motivação fez  
este trabalho comigo.

## **AGRADECIMENTOS**

Agradeço a meus orientadores, sem os quais este trabalho não seria possível;  
Aos membros do SpaceLab, em especial a equipe do FloripaSat-2 pela ajuda e suporte;  
À minha família, especialmente minha mãe, que celebrou, reclamou, agradeceu e chorou comigo durante toda a graduação; À todos os servidores da UFSC, por manter e prover educação pública, gratuita e de qualidade;

*“O êxito precoce é um péssimo professor. Quando isso acontece, somos recompensados por nossa falta de preparação e, quando nos vemos no meio de uma situação para a qual devemos nos preparar, não somos capazes. Não sabemos como fazê-lo.”*  
(Chris Hadfield, *An Astronaut's Guide to Life on Earth*, 2013)

## RESUMO

Satélites artificiais são projetos que demandam níveis elevados de confiabilidade de seus módulos. Apesar de alguns projetos permitirem atualizações posteriores a firmware e programas, são raros os casos onde é possível revisar e corrigir hardware. Por esse motivo a etapa de testes e AIV (*Assembly, Integration, and Verification*) é crucial para a garantia de que o projeto não corra mais riscos do que os que são inerentes à área. Esses fatores são amplificados quando tratamos de *CubeSats*, que possuem escopo e orçamentos menores quando comparados a grandes projetos governamentais e/ou comerciais. Este trabalho propõe a implementação de um sistema de *workflows* hospedados na plataforma de controle de versionamento *GitHub* que, aliados à funcionalidade *GitHub Actions*, permitirá a execução automatizada de testes no contexto dos planos de *Assembly, Integration, and Verification*(AIV) da missão *FloripaSat-2* e, posteriormente, análise e interpretação dos dados coletados de modo a obter resultados qualitativos e quantitativos da execução.

**Palavras-chave:** CubeSat. Nanossatélite. FloripaSat-2. AIV.

## ABSTRACT

Artificial Satellites are projects that demand highly reliable modules. Despite some missions allowing post-deployment updates to firmware and software, cases where it is possible to perform hardware maintenance are rare. Because of this fact, the *Assembly, Integration and Verification* (AIV) process is crucial to guarantee that the project doesn't face more risks than those which are inherent of a space mission. These factors are intensified when dealing with *CubeSats*, that statistically have lower budgets and scope when compared to large scale governmental and/or commercial space missions. This article proposes the implementation of a test automation workflow system hosted at GitHub that will make use of the GitHub Actions tool to allow automated execution of tests during the AIV step of the FloripaSat-2 mission, and lastly, will allow collection and analysis of data in order to draw relevant conclusions regarding the execution.

**Keywords:** CubeSat. Nanosatellite. FloripaSat-2. AIV.

## LISTA DE FIGURAS

Figura 1 – Renderização do FloripaSat-2. . . . .	13
Figura 2 – Lançamentos anuais de Nanossatélites (previsão) . . . . .	22
Figura 3 – Estações que receberam dados enviados pelo FloripaSat-I nos três primeiros meses da missão. . . . .	25
Figura 4 – Tela de resultados após a execução de um <i>workflow</i> . . . . .	39
Figura 5 – Fluxo de execução do modelo de automação . . . . .	43



## LISTA DE QUADROS

Quadro 1 – Termos e definições na disciplina de testes de <i>software</i> . . . . .	17
Quadro 2 – Pesquisas realizadas . . . . .	19

## LISTA DE TABELAS

Tabela 1 – Tabela comparativa de tempos de execução em segundos do workflow de testes unitários dos <i>drivers</i> do OBDH 2.0 . . . . .	40
Tabela 2 – Tabela comparativa de tempos de execução em segundos do workflow de testes unitários dos <i>devices</i> do OBDH 2.0 . . . . .	40
Tabela 3 – Execuções do workflow drivers . . . . .	60
Tabela 4 – Execuções do workflow . . . . .	60

## **LISTA DE ABREVIATURAS E SIGLAS**

AIV	<i>Assembly, Integration, and Verification</i>
CSA	<i>Canadian Space Agency</i>
FlatSat	Plataforma de testes para módulos CubeSat
NASA	<i>National Aeronautics and Space Administration</i>
SpaceLab	Laboratório de Pesquisa em Tecnologias Espaciais - UFSC
UFSC	Universidade Federal de Santa Catarina

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b>	<b>13</b>
1.1	OBJETIVOS	14
1.1.1	Objetivo Geral	14
1.1.2	Objetivos Específicos	15
1.2	ORGANIZAÇÃO DO TRABALHO	15
<b>2</b>	<b>FUNDAMENTAÇÃO TEÓRICA</b>	<b>16</b>
2.1	CUBESAT	16
2.2	TESTE DE SOFTWARE	16
2.3	AUTOMAÇÃO DE TESTES	17
2.4	METODOLOGIA	18
<b>3</b>	<b>TRABALHOS RELACIONADOS</b>	<b>19</b>
3.1	SOFTWARE RELIABILITY GROWTH WITH TEST COVERAGE (MALAIYA <i>et al.</i> , 2002)	19
3.2	HOW DO SOFTWARE DEVELOPERS USE GITHUB ACTIONS TO AUTOMATE THEIR WORKFLOWS? (KINSMAN <i>et al.</i> , 2021)	20
3.3	AN ARCHITECTURE-TRACKING APPROACH TO EVALUATE A MODULAR AND EXTENSIBLE FLIGHT SOFTWARE FOR CUBESAT NANOSATELLITES (GONZALEZ <i>et al.</i> , 2019)	20
3.4	INTEGRATION AND VERIFICATION APPROACH OF ISTSAT-1 CUBESAT (MONTEIRO <i>et al.</i> , 2019)	21
3.5	QUALIFICATION AND VALIDATION TEST METHODOLOGY OF THE OPEN-SOURCE CUBESAT FLORIPASAT-I (MARCELINO <i>et al.</i> , 2020)	22
3.6	A CRITICAL EMBEDDED SYSTEM CHALLENGE: THE FLORIPASAT-1 MISSION (MARCELINO <i>et al.</i> , 2020)	23
3.7	IN-ORBIT PRELIMINARY RESULTS FROM THE OPEN-SOURCE EDUCATIONAL NANOSATELLITE FLORIPASAT-I (MARCELINO <i>et al.</i> , 2021)	24
3.8	CONSIDERAÇÕES	25
<b>4</b>	<b>FLUXO DE AUTOMAÇÃO DE TESTES</b>	<b>27</b>
4.1	VISÃO GERAL DO SISTEMA	27
4.2	O MODELO	28
4.3	FUNCIONAMENTO	28
4.4	PADRÃO DE PROJETO	29
4.5	CONFIABILIDADE	30

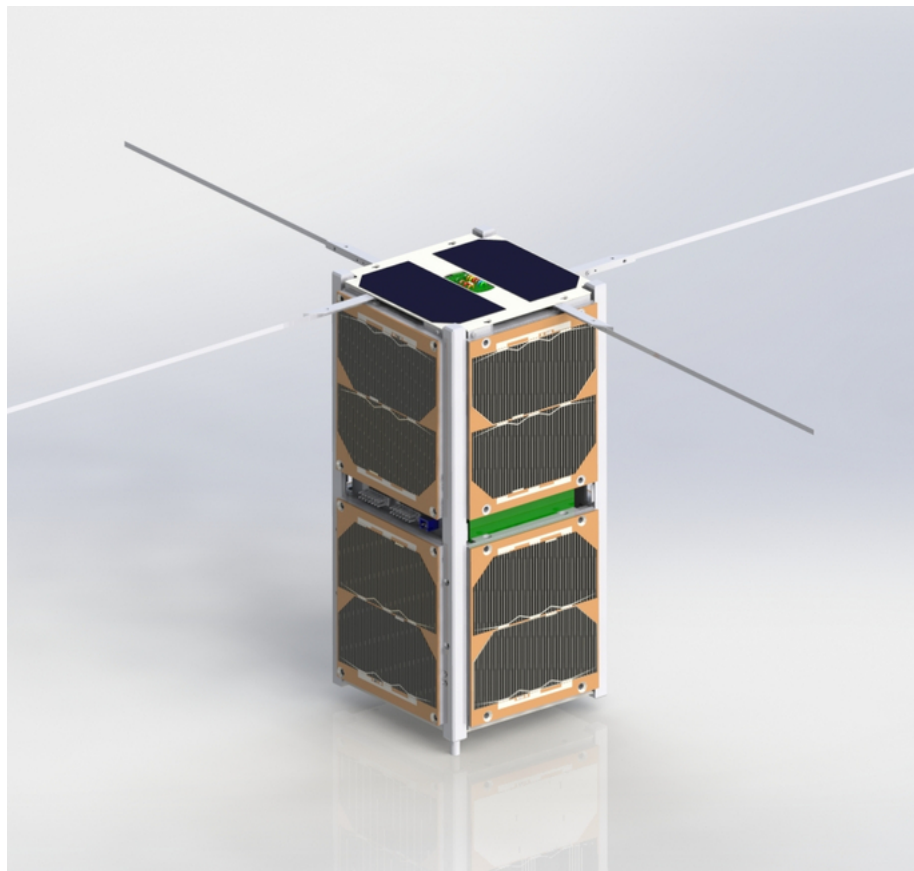
4.6	TESTES DE UNIDADE . . . . .	30
4.7	ANÁLISE ESTÁTICA . . . . .	31
<b>5</b>	<b>IMPLEMENTAÇÃO DO MODELO . . . . .</b>	<b>32</b>
5.1	FLORIPASAT-2 . . . . .	32
<b>5.1.1</b>	<b>EPS 2.0 . . . . .</b>	<b>32</b>
<b>5.1.2</b>	<b>TTC 2.0 . . . . .</b>	<b>33</b>
<b>5.1.3</b>	<b>OBDH 2.0 . . . . .</b>	<b>33</b>
5.2	ANÁLISE ESTÁTICA . . . . .	34
5.3	TESTES DE UNIDADE . . . . .	34
<b>5.3.1</b>	<b>CMocka . . . . .</b>	<b>34</b>
5.3.1.1	Um <i>Mock Object</i> na CMocka . . . . .	35
5.4	AUTOMAÇÃO . . . . .	38
<b>5.4.1</b>	<b>GitHub Actions . . . . .</b>	<b>38</b>
<b>5.4.2</b>	<b>Pré Execução . . . . .</b>	<b>39</b>
<b>5.4.3</b>	<b>Execução . . . . .</b>	<b>42</b>
<b>6</b>	<b>RESULTADOS . . . . .</b>	<b>44</b>
6.1	EXECUÇÕES . . . . .	44
6.2	BUGS ENCONTRADOS . . . . .	44
6.3	CONSIDERAÇÕES . . . . .	44
<b>7</b>	<b>CONSIDERAÇÕES FINAIS . . . . .</b>	<b>45</b>
7.1	TRABALHOS FUTUROS . . . . .	46
<b>7.1.1</b>	<b>Testes de Integração . . . . .</b>	<b>46</b>
<b>7.1.2</b>	<b>FlatSat . . . . .</b>	<b>46</b>
<b>7.1.3</b>	<b>Containerização . . . . .</b>	<b>46</b>
	<b>REFERÊNCIAS . . . . .</b>	<b>47</b>
	<b>APÊNDICE A – WORKFLOW DE TESTES DE UNIDADE PARA O</b>	
	<b>MÓDULO OBDH 2.0 . . . . .</b>	<b>52</b>
A.1	UNIT-TESTS-DEVICES.YML . . . . .	52
	<b>APÊNDICE B – DEPLOY JSON . . . . .</b>	<b>55</b>
B.1	DEPLOYJSON.PY . . . . .	55
	<b>APÊNDICE C – ADS1248_WRAP.C . . . . .</b>	<b>57</b>
	<b>APÊNDICE D – EXECUÇÕES DOS DRIVERS WORKFLOWS DO</b>	
	<b>OBDH2 . . . . .</b>	<b>60</b>
	<b>ANEXO A – DEVICE ADC . . . . .</b>	<b>62</b>

## 1 INTRODUÇÃO

CubeSats são uma classe de artefatos espaciais elaborada com o intuito de reduzir custos e tempo de desenvolvimento, além de providenciar maior acessibilidade ao espaço (JOHNSTONE, 2020). Inicialmente projetados para utilização educacional em universidades (BURT, 2011), são amplamente usados para exploração espacial em órbita terrestre baixa, com altitudes entre 160km e 2000km (ALANAZI; STRAUB, 2019).

CubeSats possuem uma unidade básica (U) de dimensão 10cm x 10cm x 10cm e de até 1kg de massa, mas podem ser configurados em até 24 Unidades (ou 24U) (CANADIAN SPACE AGENCY - CSA, 2018). O FloripaSat-2 (SPACELAB - UFSC, 2021), projeto do Laboratório de Pesquisa em Tecnologias Espaciais da UFSC no qual este trabalho se baseia, é um CubeSat 2U que se encontra no presente momento, em estágio de desenvolvimento ativo. A figura 1 apresenta uma renderização da configuração do FloripaSat-2.

Figura 1 – Renderização do FloripaSat-2.



Fonte: (SPACELAB - UFSC, 2021)

O processo de verificação de design é uma etapa crucial em projetos de engenharia. Em tradução livre dos autores, Monteiro et al, dizem que:

Em projetos espaciais, a amplitude e cuidado minucioso dos processos de teste são ainda mais importantes, em vista do nível de confiabilidade a ser imposto no produto final, que é comumente um artefato espacial que precisa resistir à uma gama de ameaças e funcionar em um ambientes inóspitos. (MONTEIRO *et al.*, 2019)

Ainda segundo os autores, o processo de *A/V* (sigla em inglês para Montagem, Integração, e Verificação) tende a ser mais leve em CubeSats, devido à menor escala dos projetos.

No entanto, o risco de um projeto de nanossatélite não é desprezível. Em um estudo realizado sobre os lançamentos de CubeSats entre 2003 e 2015, os autores (PANGA *et al.*, 2016) constataram que cerca de 15% das missões CubeSat nesse período não conseguiram manter comunicações com as estações de controle, resultando em falha da missão após o lançamento.

Por esses motivos, percebe-se então a relevância de um plano de testes bem estruturado para o ciclo de desenvolvimento do FloripaSat-2. Através da elaboração e execução de testes, pretende-se elevar a confiabilidade (CHEN *et al.*, 2001) do sistema, a fim de diminuir a probabilidade de que problemas no satélite resultem em falha da missão

O trabalho propõe, então, dois pontos principais:

- A implementação de um sistema de workflows hospedados na plataforma GitHub Actions (GITHUB, s.d.) nos repositórios oficiais do projeto, que possibilite a execução automática de testes unitários nos sistemas do FloripaSat-2;
- A análise descritiva dos testes unitários implementados durante o desenvolvimento dos subsistemas do satélite.

Os resultados esperados deste trabalho são a criação de um modelo para automação a ser usado em futuros projetos de desenvolvimento de sistemas embarcados para CubeSats, e um relatório com as conclusões e análises obtidas por meio do estudo dos dados coletados.

## 1.1 OBJETIVOS

### 1.1.1 Objetivo Geral

O objetivo principal do trabalho é implementar um sistema de automação dos testes unitários do firmware do FloripaSat-2 hospedado nos repositórios oficiais no *GitHub*. Esses *workflows* serão empregados durante o ciclo de desenvolvimento da

missão, executando os testes de forma automática seguindo os critérios de execução estabelecidos pelo projeto.

### 1.1.2 Objetivos Específicos

1. Apresentar uma introdução ao estado da arte referente a testes de software, desenvolvimento de CubeSats e relacionar os dois temas no contexto do FloripaSat-2;
2. Implementar *workflows* capazes de automatizar a execução de testes unitários;
3. Apresentar um estudo sobre os testes dos sistemas do FloripaSat-2 que foram implementados durante o andamento do projeto;
4. Disponibilizar códigos-fonte, dados de registro e resultados obtidos.

## 1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado do seguinte modo:

- O capítulo 2 apresenta a definição da metodologia utilizada e a formalização da fundamentação teórica utilizada na elaboração do trabalho;
- O capítulo 3 apresenta uma breve revisão de alguns trabalhos relacionados que abordam o desenvolvimento e testes de CubeSats;
- O capítulo 4 apresenta e descreve o sistema proposto;
- O capítulo 5 apresenta a descrição formal da implementação do sistema
- O capítulo 6 apresenta uma análise dos resultados observados
- O capítulo 7 traz as conclusões e considerações sobre o trabalho, e explora possíveis trabalhos futuros



## 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação teórica utilizada na elaboração do trabalho, trazendo conceitos básicos de nanossatélites e automação de testes, além de um aprofundamento sobre desenvolvimento e implementação dos mesmos.

### 2.1 CUBESAT

CubeSats são uma categoria de satélites miniaturizados cuja principal característica é possuir dimensões e massa reduzidos. De acordo com o documento oficial de padrões de design (JOHNSTONE, 2020), CubeSats devem ter estrutura em formato de cubo de lado 100mm e não podem exceder 1kg de massa.

Essas características correspondem a uma unidade, ou um CubeSat 1U. CubeSats podem ser empilhados em múltiplas unidades, em configurações como 2U, 3U, 6U ou 12U.

O padrão CubeSat foi desenvolvido pela *California Polytechnic State University*, e é uma especificação de projeto de colaboração internacional entre governos, universidades, escolas e o setor privado, com o principal objetivo de conceder acesso ao espaço para cargas pequenas, e possibilitar missões com objetivos diversos, como demonstrações de tecnologia, como foi o caso da missão Mars Cube One, desenvolvida pelo Jet Propulsion Laboratory e enviada à Marte (JPL, 2018), ou com objetivos educacionais, como os CubeSats desenvolvidos pelo SpaceLab da UFSC.

Os CubeSats tradicionalmente possuem custo e risco baixos. Por serem fabricados em sua maioria com peças e materiais provenientes de indústrias convencionais, o orçamento de uma missão não se compara aos orçamentos bilionários de missões convencionais. E do mesmo modo, o risco de se perder um CubeSat em uma missão que resulte em falha não é tão impactante em termos orçamentários e de trabalho perdido. Esse fato faz com que esse tipo de missão seja ideal para ser utilizado com propósitos educativos por instituições de ensino. A missão FloripaSat-1 da UFSC, por exemplo, teve como um dos objetivos treinar e capacitar estudantes de engenharia no processo de desenvolvimento e operação de uma missão espacial (MARCELINO *et al.*, 2020).

### 2.2 TESTE DE SOFTWARE

A disciplina de testes de *software*, segundo Raul Wazlawick, se tornou extremamente importante no ciclo de desenvolvimento, sendo considerada por modelos ágeis como atividade crítica, muitas vezes determinando que os testes devem ser escritos antes mesmo da implementação dos componentes (WAZLAWICK, 2019).

Ainda segundo Wazlawick, quando se trata de teste é importante definir termos

que, apesar de parecerem sinônimos, tem significados distintos na área de testes de software, como apresentados no Quadro 1

Quadro 1 – Termos e definições na disciplina de testes de *software*

Termo	Definição
Erro ( <i>error</i> )	Diferença detectada entre o resultado de uma computação e o resultado correto esperado
Defeito ( <i>fault</i> )	Linha de código, bloco ou conjunto de dados incorretos que provocam erros
Falha ( <i>failure</i> )	Não funcionamento do <i>software</i> , possivelmente causada por algum defeito
Engano ( <i>mistake</i> )	Ação que produz um defeito no <i>software</i>

Fonte: (WAZLAWICK, 2019)

Ainda outra distinção a ser feita é entre verificação, validação e teste:

Verificação: consiste em analisar o *software* para ver se ele está sendo construído de acordo com o que foi especificado;

Validação: consiste em analisar o *software* construído para ver se ele atende às verdadeiras necessidades dos interessados;

Teste: é uma atividade que permite realizar a verificação e a validação do *software*.

(WAZLAWICK, 2019)

A disciplina de testes de *software* abrange múltiplas áreas de interesse, e objetivo de testes, que variam desde testes de segurança e tolerância a falhas, até integração de sistema. Os testes empregados e utilizados como base para o estudo deste trabalho, são os testes de unidade, que são considerados os testes básicos, e tem o objetivo de verificar de maneira isolada o funcionamento correto de componentes do código. No FloripaSat-II, os testes de unidade foram implementados como descritos na Seção 4.6 e Seção 5.3.

## 2.3 AUTOMAÇÃO DE TESTES

Durante o tempo de vida de um projeto de um software ou um sistema de softwares, como é o caso de um projeto de satélites artificiais, os componentes envolvidos invariavelmente passarão por algumas mudanças. Mudanças essas que podem introduzir novos *bugs* em componentes que anteriormente funcionavam. A automação de testes, como definido pelos autores do livro *Software Test Automation - Effective use of test execution tools*, se dá pelo emprego de tecnologias que permitam que um caso de teste seja executado de maneira autônoma, sem intervenção ou monitoramento, de maneira muito mais eficiente (FEWSTER; GRAHAM, 1999) do que se executado via interação humana.

A etapa de testes é essencial em qualquer projeto de engenharia, e pode-se dizer que é ainda mais importante em projetos espaciais, já que erros de projeto, sejam eles de hardware ou firmware, podem resultar em falha da missão.

Este trabalho propõe adotar a ferramenta de automação de testes em uma missão CubeSat, mais especificamente a FloripaSat-2, desenvolvida pelo laboratório SpaceLab da Universidade Federal de Santa Catarina.

A missão FloripaSat-2 utiliza os testes de software implementados para validar os seus subsistemas.

## 2.4 METODOLOGIA

O trabalho propõe a implementação de um modelo de automação dos Testes de Unidade desenvolvidos para os subsistemas do FloripaSat-II, e apresenta detalhes sobre a motivação, construção e desenvolvimento desse modelo. Também é realizado um estudo e análise da implementação dos testes de unidade, onde é demonstrada a maneira como foram desenvolvidos, quais as tecnologias que permitem sua funcionalidade, e qual o seu papel no ciclo de desenvolvimento da missão.

Para isso, as ferramentas utilizadas são descritas em detalhes técnicos de funcionamento e finalidade, e são apresentados trechos de código desenvolvidos nas seções em que referências diretas às implementações se fazem úteis para a compreensão do projeto.

### 3 TRABALHOS RELACIONADOS

No motor de pesquisa acadêmica Google Scholar foram feitas pesquisas associando as áreas de conhecimento de desenvolvimento de nanossatélites, testes de software e automação de testes. As pesquisas foram realizadas ao longo da elaboração do trabalho e diferentes *strings* de busca foram utilizadas. Como critério para escolha das publicações à serem estudadas, foram analisados os resultados apresentados na primeira página do motor de busca. Para determinar a relevância de uma publicação para o trabalho, foi analisado o *abstract* e a introdução dos mesmos. Para os trabalhos relacionados apresentados neste capítulo os resultados e conclusões também foram estudados.

Quadro 2 – Pesquisas realizadas

<i>String</i>	Resultados
software+reliability+test	565,000
nanosatellite + design	15,200
software + test	4,300,000
software+test+automation	1,970,000
flatsat + cubesat	359
cubesat	29,200
flatsat + nanosatellite	273
cubesat + testbed	3,510

Fonte: Autor.

#### 3.1 SOFTWARE RELIABILITY GROWTH WITH TEST COVERAGE (MALAIYA *et al.*, 2002)

O termo "cobertura de testes", do inglês *test coverage* quantifica o grau de rigor dos testes. Existem ferramentas que são capazes de mensurar a cobertura de testes em diferentes termos, como *branches*, e integração de uso. Os autores deste artigo trazem uma relação entre duração dos testes, cobertura e confiabilidade de software.

Desenvolvedores são capazes de atingir o nível de confiabilidade de *software* de maneira previsível a partir de análise de confiabilidade durante o desenvolvimento. Para quantificar essa métrica durante os testes, o código é executado com *inputs* selecionados aleatoriamente seguindo uma distribuição determinada. Dessa forma, se a distribuição utilizada for a mesma esperada durante a vida operacional do *software*, pode ser levantado um modelo de confiabilidade a partir dos resultados dos testes.

Os autores constatarem, no entanto, que para focar os testes na detecção de erros, é mais produtivo executar o código com *inputs* considerados especialmente para estimular comportamentos onde erros são mais prováveis, e propõem neste estudo um modelo para realizar essa mensuração. Dentre os resultados obtidos na

análise, os autores constatam que em muitos casos, um mínimo de 85% de cobertura de testes é necessário para aumento significativo na confiabilidade do software em desenvolvimento.

### 3.2 HOW DO SOFTWARE DEVELOPERS USE GITHUB ACTIONS TO AUTOMATE THEIR WORKFLOWS? (KINSMAN *et al.*, 2021)

Ferramentas sociais de programação, como o GitHub, alteraram a natureza colaborativa de desenvolvimento de *software open source*, proporcionando novas oportunidades para engajamento, mas ao mesmo tempo aumentando a carga de trabalho de mantenedores de repositórios, que devem revisar, comunicar, explicar o projeto e gerenciar o versionamento de um número muito maior de colaboradores.

Para reduzir essa carga de trabalho intensiva, os autores estudam a ferramenta GitHub Actions, disponibilizada em 2019, que permite aos usuários automatizar questões como verificar se o código compila, se todos os testes são executados com sucesso, ou ainda se o código está dentro dos padrões de desenvolvimento do projeto. Esse controle pode ser feito segundo diversos *triggers*, como *commits*, comentários, *issues* ou *pull-requests*.

No artigo, os autores estudaram como desenvolvedores de software fazem uso da ferramenta para automatizar os fluxos de trabalho, como a ferramenta é discutida, e quais seus efeitos nos *pull requests* dos projetos. Com uma amostra de 3,190 repositórios ativos no GitHub, e análise estatística de 926 deles, foi constatado que apenas uma pequena parcela dos repositórios utiliza a ferramenta, mas que a plataforma é vista, em geral, como benéfica para os contribuidores. Entre esses repositórios, os autores encontraram 708 *workflows* de automação únicos.

Os autores relatam também que a utilização da ferramenta acarretou em diminuição da quantidade de aprovação de *commits* mensal, e aumentou a quantidade de *pull requests* rejeitados, experiências anedóticas revelam que a ferramenta e as funcionalidades que ela apresenta são recebidas com entusiasmo e apreciação pelos desenvolvedores.

### 3.3 AN ARCHITECTURE-TRACKING APPROACH TO EVALUATE A MODULAR AND EXTENSIBLE FLIGHT SOFTWARE FOR CUBESAT NANOSATELLITES (GONZALEZ *et al.*, 2019)

Um ponto importante para aumentar o sucesso de missões CubeSat é a entrega de *software* de voo de qualidade. Esse tópico é identificado como chave para elevar a qualidade de diversos pontos de uma missão, como reusabilidade, colaboração, redução de riscos e facilitação de desenvolvimento. Uma arquitetura *software* de voo apropriada representa os requisitos funcionais e não funcionais e guia o desenvolvi-

mento. Dessa forma, para atingir o nível de qualidade desejada a arquitetura deve ser monitorada durante todo o ciclo de desenvolvimento de *software*. Esse monitoramento, no entanto, é difícil e não-trivial.

Neste artigo, Gonzales *et al.* apresentam detalhes sobre a implementação de missões CubeSat que se mostraram bastante similares à metodologia empregada no desenvolvimento do FloripaSat-2. Os autores trazem em seu trabalho pontos importantes sobre como entregar software de qualidade, confiável e tolerante a falhas para missões CubeSat, que como toda missão espacial, se tratam de sistemas críticos que de modo geral devem operar de maneira autônoma e cujo software e hardware não podem ser corrigidos após o lançamento.

Os pontos de maior relevância para este trabalho são os requisitos não-funcionais de uma missão CubeSat trazidos pelos autores, entre eles a confiabilidade e extensibilidade do código, e a metodologia de projeto e arquitetura do *software* de voo, como a distinção entre as camadas do sistema, utilização de repositórios e em especial os métodos de validação utilizando métodos de integração contínua como ferramenta para elevar a confiabilidade do sistema através de testes automatizados.

Os temas apresentados pelos autores serão discutidos de forma mais profunda na Seção 4.5 e Seção 5.1.

### 3.4 INTEGRATION AND VERIFICATION APPROACH OF ISTSAT-1 CUBESAT (MONTEIRO *et al.*, 2019)

Satélites artificiais precisam resistir a uma grande quantidade de adversidades enquanto operam em um ambiente hostil e inóspito. Por isso é essencial que sejam feitos testes extensivos para garantir o nível de confiança necessário para que o objetivos da missão sejam atingidos. Porém, missões de CubeSat tendem a não ser tão rigorosas em suas etapas de montagem, integração e verificação (AIV), fato que é refletido na taxa de missões que acabam em falhas.

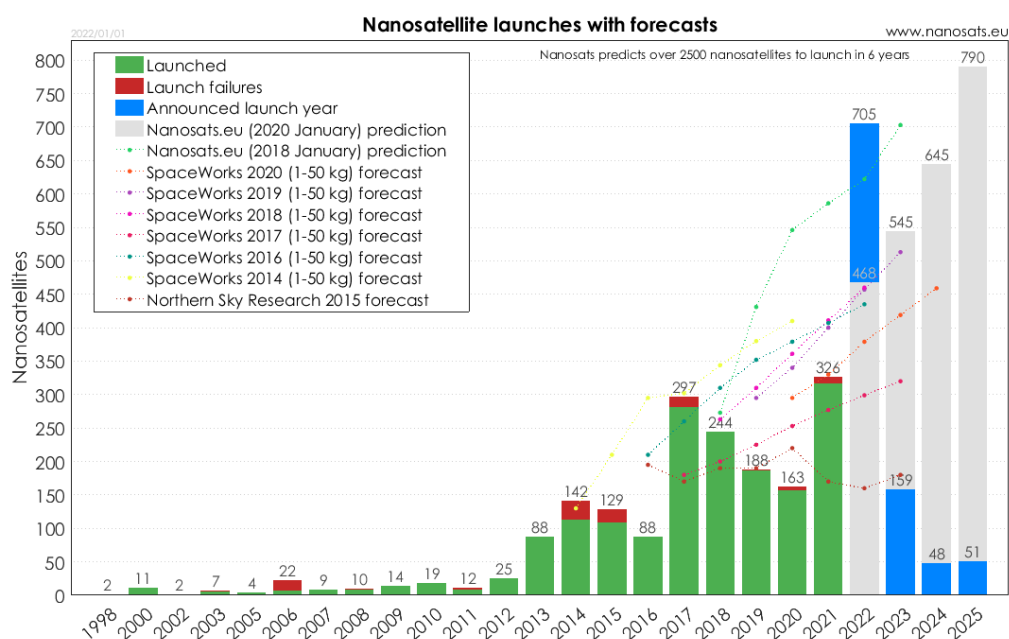
Esse paradigma, no entanto, está mudando a medida que a importância do CubeSats aumenta. Ela vai de simplesmente projetos educacionais, para missões de custos e importância elevados. Um exemplo desse salto é a missão Mars Cube One, ou MarCO (JPL, 2018), desenvolvida pelo *Jet Propulsion Laboratory* da Nasa, que enviou dois CubeSats a Marte, para servirem como satélites de comunicação. Assim, os autores desse artigo apresentam a ideia de que integração e verificação em missões CubeSats passam a ser encaradas com mais seriedade, como demonstrado na figura 2, onde é possível observar o aumento de lançamento de nanossatélites e, paralelamente, a diminuição da taxa de falhas.

Como estudo de caso, o artigo apresenta o processo de AIV empregado no desenvolvimento do ISTSat-1 (ISTSAT-1... , 2021), o primeiro nanossatélite português, lançado em 2021 e que utilizou a estratégia de teste através de uma *FlatSat*, que se

trata de uma plataforma de testes de *Hardware In the Loop* (HIL).

A equipe do ISTSat-1, segundo os autores, optou por desenvolver a maioria dos subsistemas do satélite e adotou métodos ágeis para as atividades de AIV de forma a diminuir os riscos relacionados a falta de experiência da equipe. Dessa forma foi possível desenvolver e testar os módulos de forma contínua e iterativa, o que possibilitou a descoberta rápida de diversos erros e problemas que necessitavam de ajustes de design. Foi concluído nesse estudo que, caso a equipe tivesse optado por um processo de desenvolvimento em cascata, como tradicionalmente empregado em missões CubeSat, seria mais difícil, senão impossível, ter descoberto os mesmos problemas no design e funcionamento. Concluem então, que a adoção de processos iterativos é útil para projetos educacionais que contam com equipes sem muita experiência, ou sem muitos recursos financeiros.

Figura 2 – Lançamentos anuais de Nanossatélites (previsão)



Fonte: Nanosats Database (KULU, ERIK, 2021)

### 3.5 QUALIFICATION AND VALIDATION TEST METHODOLOGY OF THE OPEN-SOURCE CUBESAT FLORIPASAT-I ("MARCELINO *et al.*, 2020)

Esta seção e a próxima apresentam dois trabalhos produzidos como resultado da missão FloripaSat-1, uma missão de demonstração desenvolvida inteiramente por estudantes da Universidade Federal de Santa Catarina (UFSC), cujo CubeSat foi lançado em 2019.

O FloripaSat-1 é composto de três módulos distintos, o sistema elétrico - EPS (*Electric Power System*), o módulo de gerência de dados - OBDH (*Onboard Data*

*Handling*) e o módulo de telemetria e telecomandos - TT&C (*Telemetry, Tracking and Command*).

O artigo informa que a missão tinha como objetivo testar tecnologias que possibilitem o desenvolvimento rápido e de baixo custo de satélites espaciais, além de treinar estudantes nas áreas de concepção, implementação e operação de uma missão espacial completa. Além disso, o FloripaSat-1 é um projeto *open source* e as informações de software e hardware dos módulos desenvolvidos estão disponíveis em repositórios públicos, para uso em futuras missões.

Os autores descrevem que o FloripaSat-1 foi desenvolvido com base em projetos de engenharia de sistemas, dividido em Modelos de protótipo (PM), engenharia (EM-I e EM-II) e finalmente o Modelo de Vão (FM). Foram realizados testes em cada uma dessas etapas para validar os sistemas e os resultados, segundo o artigo, foram decisivos na detecção e correção de erros.

Apesar do artigo focar em integração e validação dos módulos a nível de hardware, julga-se relevante para a elaboração deste trabalho, que foca principalmente em software, por descrever as diferentes condições que uma missão espacial é submetida, além da metodologia de desenvolvimento adotada pela equipe do FloripaSat-1, em que a grande maioria dos membros seguiu para a missão FloripaSat-2, a qual este trabalho se baseia.

### 3.6 A CRITICAL EMBEDDED SYSTEM CHALLENGE: THE FLORIPASAT-1 MISSION (MARCELINO *et al.*, 2020)

Este artigo, elaborado também a partir da missão FloripaSat-1 do SpaceLab, apresenta definições e descrições dos subsistemas do satélite, além de algumas informações e resultados de testes e simulações realizadas nos mesmos.

Como descrito anteriormente, o FloripaSat-1 possui três sub sistemas principais:

- *Electric Power System* (EPS)
- *On-Board Data Handling* (OBDH)
- *Tracking, Telemetry and Command* (TT&C)

O EPS é o módulo responsável por distribuir, coletar e armazenar a energia utilizada pelo satélite. A energia é coletada através de painéis solares e armazenada em uma bateria de íon de lítio. A distribuição da energia é definida a partir de um microcontrolador que analisa os estados de carga e de energia e decide quais módulos permanecerão em operação.

O OBDH é o módulo responsável pela gerência de atividades do satélite. Ele realiza a interface entre todos os subsistemas do satélite. Os dados gerados são empacotados e transmitidos através do TT&C, e dados recebidos são enviados ao OBDH



para que ele execute a tarefa requisitada, ou então envie o comando ao módulo requisitado. Este módulo também possui uma memória, para que possam ser futuramente recuperados.

Finalmente, o TT&C é o módulo responsável pela comunicação entre o satélite e as estações de controle na Terra. Ele opera através de dois módulos de rádio, um para a banda VHF e um para UHF. Os comandos a serem enviados e recebidos pelo satélite (*downlink/ uplink*) são transmitidos através do módulo UHF e a banda VHF é reservada para transmissões do tipo *beacon*.

Os autores discorrem ainda sobre os demais módulos e subsistemas do satélite, como outros *payloads* que não foram desenvolvidos ou projetados pela equipe da UFSC e portanto, ficam de fora desse breve resumo.

A relevância deste artigo se dá pelo fato de que o FloripaSat-2 utiliza a mesma estrutura de subsistemas, com os seus módulos sendo sucessores diretos do software e hardware utilizados na missão anterior, operando de forma idêntica ou muito similar.

### 3.7 IN-ORBIT PRELIMINARY RESULTS FROM THE OPEN-SOURCE EDUCATIONAL NANOSATELLITE FLORIPASAT-I

(MARCELINO *et al.*, 2021)

O FloripaSat-I, como mencionado anteriormente, constituiu uma missão CubeSat desenvolvida na Universidade Federal de Santa Catarina, em que os objetivos incluíam testar tecnologias que permitissem desenvolvimento mais rápido e menos custoso de satélites futuros através do reuso da estrutura *open source*. Lançado em 2019 em um foguete Chinês Long March 4B, os dados transmitidos pelo satélite foram usados como métricas de validação da missão, permitindo aos desenvolvedores (em sua maioria estudantes da UFSC) traçar paralelos entre os dados de órbita e as análises obtidas em laboratório.

Os autores apresentam uma análise dos resultados coletados durante os primeiros três meses da missão, além das lições aprendidas através dos dados recebidos.

O FloripaSat-I foi lançado em uma órbita heliossíncrona, que se trata de uma configuração de órbita quase polar, onde o plano de órbita é sempre fixo em relação ao Sol, e o objeto em órbita passa pela sobre a mesma região da superfície no mesmo instante todos os dias.(TSCHERBAKOVA *et al.*, 1998).

Os dados enviados pelo satélite foram coletados em múltiplas estações espalhadas pelo mundo, como apresentado na figura Figura 3, e decodificados através de um *software* de estação terrestre desenvolvido no laboratório.

No estudo publicado, os autores elencam algumas das dificuldades encontradas durante o desenvolvimento da missão e reveladas pelos dados do satélite em órbita, em especial dificuldades na montagem, que pode ter acarretado em problemas estruturais e de funcionamento, e de frequências utilizadas, que dificultaram a comunicação.

Figura 3 – Estações que receberam dados enviados pelo FloripaSat-I nos três primeiros meses da missão.



Fonte: (MARCELINO *et al.*, 2021)

Além disso, foi realizada uma análise com os participantes do projeto com o intuito de compreender como a missão os afetou, e os resultados obtidos foram em geral percebidos como positivos pelos estudantes, que reportaram não apenas evolução de aprendizado e de experiência de desenvolvimento.

### 3.8 CONSIDERAÇÕES

Os artigos e teses escolhidos para embasar e fundamentar este trabalho estabelecem a importância de um modelo de testes bem estruturado em um projeto de desenvolvimento de *software*. Todos os trabalhos relacionados a missões CubeSat mencionam a importância dos testes para a descoberta e solução de erros, bem como a relação entre a execução e planejamento dos testes e o aumento de confiabilidade do *software*., como demonstrado pelo estudo apresentado na Seção 3.1.

O artigo apresentado na Seção 3.3 é especialmente interessante para a proposta desse trabalho por descrever em detalhes o processo e as práticas de desenvolvimento aplicadas na série de nanossatélites *SUCHAI*, que se assemelham ao modelo proposto neste trabalho. Os autores citam diversos modelos e práticas de desenvolvimento que são, em essência, seguidos de maneira muito próxima pela missão FloripaSat-II, como será evidenciado no Capítulo 4 e Capítulo 5.

O artigo analisado na Seção 3.7, apresenta uma análise dos resultados obtidos nos meses iniciais de operação do FloripaSat-I, bem como uma análise do impacto do satélite nos estudantes envolvidos no projeto. Este artigo demonstra as dificuldades encontradas no desenvolvimento da missão, e servem o propósito de indicar pontos

de atenção e melhoria para trabalhos futuros, como é o caso do FloripaSat-II, que se inspira e por vezes diretamente reimplementa os sistemas utilizados no seu antecessor.

Além disso, os outros dois trabalhos produzidos por membros do SpaceLab, descritos na Seção 3.6 e Seção 3.5, que relatam o processo de desenvolvimento da missão FloripaSat-1 mostram a importância de CubeSats como ferramenta educacional, servindo para treinar e capacitar estudantes no projeto, desenvolvimento e operação de uma missão espacial completa.

O modelo proposto por este trabalho é, então, uma continuidade dos pensamentos apresentados neste capítulo. Espera-se que o sistema implementado permita aumentar a qualidade e a confiabilidade do código entregue através de técnicas de integração contínua para automatizar os testes de unidade dos sistemas embarcados desenvolvidos para o FloripaSat-II, além de possibilitar mais uma ferramenta com potencial de estudo e aprendizado, sob a qual podem ser empregadas futuras ferramentas para analisar e expandir o modelo, como detalhado na Seção 7.1.

## 4 FLUXO DE AUTOMAÇÃO DE TESTES

Este capítulo descreve e formaliza o sistema proposto. A motivação para este sistema surgiu a partir de reuniões e conversas com as equipes de desenvolvimento do FloripaSat-2 no SpaceLab, onde foi decidido adotar um modelo de testes que utilize técnicas de integração contínua para automatizar o processo de publicar e testar os subsistemas desenvolvidos para o FloripaSat-2

O sistema proposto leva em consideração o ambiente de desenvolvimento, os objetivos da missão e as características do projeto para desenvolver um modelo cuja finalidade é facilitar e agilizar o ciclo de desenvolvimento, simultaneamente elevando a confiabilidade dos sistemas ao aumentar a cobertura de teste, como constatado por (CHEN *et al.*, 2001), cujo estudo apontou a correlação entre cobertura de teste e confiabilidade do software, como detalhado na Seção 4.5

### 4.1 VISÃO GERAL DO SISTEMA

O sistema proposto proporciona um ambiente de desenvolvimento onde os testes de software são executados de maneira automática e descentralizada, nos repositórios da missão FloripaSat-2. Essa abordagem proporciona, a princípio, dois pontos importantes:

- A garantia de que toda alteração, adição ou remoção de código seja testada com a mesma precisão;
- Um ambiente transparente e universal para expor os testes executados e seus resultados.

A missão FloripaSat-2 possui três subsistemas próprios, desenvolvidos inteiramente no laboratório. Esses subsistemas estão descritos com mais detalhes na seção 5.1. Por esse motivo foi decidido durante o planejamento do sistema que seria necessário implementar um modelo que pudesse ser reproduzido de maneira simples e rápida em outros repositórios, para que todo código desenvolvido para a missão fosse tratado da mesma maneira. Para isso, foi implementado uma convenção de hierarquia de arquivos e nomenclatura, de modo a garantir a reusabilidade do modelo proposto, sem que seja preciso realizar grandes alterações. Como consequência, esse padrão acarreta em maior consistência entre os subsistemas do satélite. O padrão proposto está descrito em detalhes na Seção 4.4. No entanto, ele será melhor compreendido uma vez que se conheça o contexto de implementação. Por esse motivo, as próximas seções apresentam detalhes sobre funcionamento e implementação do modelo proposto.

## 4.2 O MODELO

A implementação dos subsistemas do FloripaSat-2 é hospedada nos repositórios da missão no GitHub. Por esse motivo foi decidido utilizar a ferramenta de CI/CD desenvolvida pela própria plataforma, a GitHub Actions(GITHUB, s.d.), que proporciona um ambiente *plug and play*, onde os *workflows* são implementados e executados na própria plataforma, sem dependências externas.

O modelo foi pensado para possibilitar testar de maneira automática os sistemas embarcados do FloripaSat-2. Os módulos passam por testes unitários e análise estática, visando identificar e corrigir falhas e defeitos (melhor discutidos na Seção 2.2), e a partir disso elevar a confiabilidade do sistema de um modo geral, como exemplificado por (CHEN *et al.*, 2001).

Considerando esses pontos, os princípios básicos do modelo proposto são:

- Testar os subsistemas de maneira automática
- Utilizar o GitHub Actions como hospedagem
- Ser facilmente transferido e implementado em outros projetos

## 4.3 FUNCIONAMENTO

O modelo proposto se trata de um fluxo de execução constituído por três ações principais: a execução de um *script* responsável por identificar e automatizar o processo; A compilação de arquivos fonte; e a execução dos arquivos compilados. Conforme elaborado na Seção 5.2, o modelo se comporta de maneira diferente no caso de execução de Análise Estática. Nesses casos, o fluxo pode ser dividido em duas etapas: a execução do *script* para identificar os arquivos fonte; e a análise estática destes mesmos arquivos.

O funcionamento detalhado do modelo é discutido em detalhes na ??, mas de maneira breve, a execução de modo geral segue a seguinte cronologia:

1. Um evento *x* dispara a execução do script, que identifica os arquivos fonte *y*;
2. Os arquivos *y* são compilados, gerando binários executáveis *z*;
3. Os binários *z* são executados, e é gerado um relatório sobre o processo ao final da execução.

Como um dos requisitos do modelo é a possibilidade de implementação em diferentes projetos, se faz necessário a padronização de alguns pontos dos projetos, de modo que as ações do fluxo de execução possam ocorrer com pouca ou nenhuma manutenção. O padrão desenvolvido está descrito na Seção 4.4 a seguir.

## 4.4 PADRÃO DE PROJETO

De modo a garantir o funcionamento correto do modelo proposto em diferentes projetos, foi necessário padronizar o modo como o *script* de automação identifica os arquivos relevantes. Para que isso aconteça, foi proposto um padrão de estrutura e nomenclatura de arquivos. Desta forma o sistema encontraria os arquivos sempre no mesmo local, e poderia a princípio ser simplesmente copiado para outros repositórios.

A título de compreensão do padrão, o trecho a seguir detalha a estruturação de um subsistema do FloripaSat-2, tomando como exemplo o commit `28c31e7a7db69240f1700ec3621436b14c517529` da branch `dev_firmware` do repositório do OBDH 2.0 (SPACELAB, 2020b). A partir da raiz do diretório, o projeto é estruturado da seguinte forma<sup>1</sup>:

- `.github/workflows`
- `doc`
- `firmware`
  - `app`
  - `config`
  - `devices`
    - \* `device 1`
    - \* `device 2`
    - \* `...`
    - \* `device n`
  - `drivers`
    - \* `driver 1`
    - \* `driver 2`
    - \* `driver ...`
    - \* `driver m`
- `hardware`
- `tests`
  - `devices`
  - `drivers`
  - `mockups`<sup>2</sup>

<sup>1</sup> alguns arquivos e diretórios foram omitidos de modo a deixar o exemplo didático.

Ao padronizar a hierarquia de arquivos do projeto, obtem-se a garantia de que os *scripts* sempre funcionarão corretamente, já que buscará os arquivos necessários sempre na mesma localização. O outro aspecto do padrão estabelecido é a nomenclatura dos arquivos de teste, tanto arquivos fonte como os binários compilados. O padrão de nomenclatura foi definido da seguinte forma:

Arquivos de teste são nomeados `<alvo>_test.c`;

Arquivos *mockup*<sup>2</sup> são nomeados `<alvo>_wrap.c`

Executáveis são nomeados `<alvo>_unit_test`

## 4.5 CONFIABILIDADE

Foi demonstrado que testes e confiabilidade são muitas vezes temas correlacionados em engenharia e projeto de software (MALAIYA *et al.*, 2002).

Missões espaciais constituem sistemas críticos, e nanossatélites CubeSat não são exceção. Eles devem operar de maneira autônoma por anos, e em muitos casos sem oportunidade de manutenção à hardware e firmware. Entre os muitos requisitos não-funcionais que devem ser considerados em missões CubeSat, os autores Gonzalez *et al.* trazem esses fatos como razão para que os sistemas empregados em satélites sejam extensivamente testados de modo a mitigar a maior quantidade possível de possibilidades de falha. No entanto, devido à natureza de uma missão espacial, "atingir cobertura de código integral é impraticável na maioria dos casos" (GONZALEZ *et al.*, 2019). Por isso, eles sugerem alguns outros modelos de desenvolvimento para reduzir pontos de falha, como por exemplo a implementação e utilização de um *datapath* claro e simples, limitar o uso de *mutexes* e alocação dinâmica de memória e, no caso do FloripaSat-2, optou-se por reduzir a utilização de numeração em ponto flutuante. A aplicação dessas *guidelines* no FloripaSat-2 são analisadas na Seção 5.1

## 4.6 TESTES DE UNIDADE

Os Testes de Unidade (do inglês *unit tests*) são, como apresentado por Raul Wazlawick:

"Os [testes] mais básicos e costumam consistir em verificar se um componente individual do *software* (unidade) foi implementado corretamente. Esse componente pode ser um método ou procedimento, uma classe completa ou, ainda, um pacote de funções ou classes de tamanho pequeno ou moderado. (WAZLAWICK, 2019).

<sup>2</sup> funcionamento e o propósito dos *mockups* são descritos na Seção 5.3.1.1

Os testes de unidade são, segundo o autor, comumente implementados pelo próprio desenvolvedor, durante a produção do código. O objetivo desse modelo, e garantem pelo menos que "todos os comandos e decisões do componente implementado tenham sido exercitados para verificar se eles tem defeitos".

Os testes de unidade implementados para os subsistemas do FloripaSat-2 seguem os princípios e propósitos apresentados acima, e o modelo proposto visa fornecer uma plataforma para que os componentes desenvolvidos sejam testados de maneira automática e consistente, sempre que uma nova alteração é publicada.

Detalhes sobre a implementação dos testes de unidade para a missão FloripaSat-2 estão apresentados na Seção 5.3, mas a princípio eles foram desenvolvidos para testar os *devices*, implementações dos componentes presentes nos subsistemas do satélite, como por exemplos sensores de temperatura e corrente, e os *drivers*, implementações de acesso ao *hardware* dos subsistemas.

## 4.7 ANÁLISE ESTÁTICA

A análise estática, como colocada pelos autores Ayewah *et. al.* (2008):

Avaliam o *software* no abstrato, sem executá-lo ou considerar um *input* específico. Ao invés de tentar provar que o código cumpre suas especificações, estas ferramentas procuram por violações de práticas razoável ou recomendadas de desenvolvimento (AYEWAH *et al.*, 2008)

Ferramentas de análise estática são úteis também, segundo os autores, para "encontrar locais onde o código possa dereferenciar um ponteiro nulo, ou onde ocorra *overflow* em *arrays*" (AYEWAH *et al.*, 2008).

A análise estática é empregada no desenvolvimento dos subsistemas do FloripaSat-2 através da ferramenta (CPPCHECK... , s.d.), que permite detectar problemas de *undefined behaviour* e construções de código perigosas. A ferramenta também analisa código que não siga a sintaxe padrão da linguagem, como é comum em sistemas embarcados. Mais detalhes sobre o emprego e o funcionamento da ferramenta de análise sintática no FloripaSat-2 está descrita na Seção 5.2.



## 5 IMPLEMENTAÇÃO DO MODELO

Este capítulo apresenta e detalha as implementações dos fluxos de execução hospedados no GitHub Actions e alguns dos testes unitários desenvolvidos, conforme mencionados no Capítulo 4.

Para contextualização do modelo, bem como compreensão do ambiente de desenvolvimento, a Seção 5.1 apresenta um detalhamento sobre a missão e como ela é dividida, em termos de desenvolvimento. Como discutido anteriormente, o FloripaSat-2 utiliza a filosofia Free Open Source Software, e portanto todos os módulos e subsistemas desenvolvidos estão publicados em repositórios abertos, no GitHub da missão. As seções 5.1.1, 5.1.2 e 5.1.3 referenciam em seu texto os repositórios oficiais de cada um dos subsistemas desenvolvidos no SpaceLab.

### 5.1 FLORIPASAT-2

Segundo Gonzales *et. al.* (2019), "o *software* de vôo de uma missão CubeSat constitui um dos elementos essenciais para garantia do sucesso da missão, já que é onde são implementados grande parte dos requisitos funcionais da missão". (GONZALEZ *et al.*, 2019). Segundos os autores, é o que também permite que missões CubeSat sejam escaláveis e reutilizáveis.

De fato, o FloripaSat-2 se beneficia destas propriedades. A missão é precedida pelo FloripaSat-1, lançado em dezembro de 2019, e como apresentado na Seção 3.6, segue a mesma metodologia e prática de desenvolvimento. Por consequencia, os módulos e substistemas, tanto *hardware* como *firmware* foram planejados para serem diretamente herdados do FloripaSat-1 ou atualizações de suas versões.

Este trabalho foi em sua maioria focado nos subsistemas EPS 2.0 e OBDH 2.0 por estarem em estágios mais avançados de desenvolvimento, e os códigos-fonte discutidos e apresentados se encontram nos repositórios oficiais e públicos da missão, em (SPACELAB, 2020a) e (SPACELAB, 2020b) respectivamente. Trechos relevantes do *firmware* são apresentados e explicados neste capítulo, e apresentados em um contexto mais amplo nos anexos e apêndices ao final do documento. Por se tratarem de projetos relativamente grandes em questão de extensão de código, eles não foram anexados integralmente nesta monografia, que se beneficia do caráter público dos projetos do SpaceLab. Toda a implementação mencionada pode ser encontrada nos repositórios do FloripaSat-2 no GitHub (SPACELAB - UFSC, 2021).

#### 5.1.1 EPS 2.0

O EPS 2 (*electrical power system*) é responsável por coletar, armazenar e distribuir energia para o FloripaSat-2. A coleta de energia é feita através de 10 painéis

solares posicionados ao longo da estrutura do satélite, e é armazenada em 4 baterias de íon de lítio. A distribuição de energia é feita através de conversores DC-AC. Tal qual o módulo desenvolvido para o FloripaSat-1 (MARCELINO *et al.*, 2020) o EPS 2.0 possui um chip DS2775G+ utilizado para controlar carga e descarga da bateria, além de monitorar alguns parâmetros como corrente e tensão. O EPS, com base nessas informações, executa suas rotinas de distribuição de energia, decidindo quais módulos e quais funcionalidades devem permanecer em operação. O EPS 2 também é capaz de controlar a temperatura das baterias, e conta com sensores de temperatura e aquecedores.

O projeto e implementação do EPS está disponível em seu repositório (SPACELAB, 2020a)

### 5.1.2 TTC 2.0

O módulo TTC 2 (Ou TT&C, *Telemetry, tracking & Telecommand*) é o módulo de comunicação do FloripaSat-2. É função dele realizar a comunicação entre o satélite e a *groundstation*. A comunicação com a estação terrestre será feita de duas maneiras, um sinal de *beacon* e um de telemetria.

O *beacon* transmite um sinal periódico contendo alguns dados básicos de identificação do satélite em conjunto com informações básicas de telemetria. O módulo de telemetria é o responsável pela comunicação de fato. Tem um módulo de comunicação bidimensional para receber telecomandos da Terra e transmitir os dados requisitados. O submódulo de telemetria é controlado por devices externos (como por exemplo o OBDH).

O projeto e a implementação do TTC 2 está disponível em seu repositório ([ttc2-github](#)).

### 5.1.3 OBDH 2.0

O OBDH 2 (*On-Board Data Handling*) é responsável pela sincronização as ações e o fluxo de dados entre os outros módulos do FloripaSat-2 e a *groundstation*. O OBDH organiza os dados em *data frames* e transmite através do TTC, ou armazena em memória não-volátil para transmissão em outro momento. Dados recebidos da *groundstation* são captados pelo TTC e direcionados para o OBDH, que realiza a ação apropriada ou transmite para o módulo apropriado.

O OBDD também é responsável por sincronizar e fazer a interface entre os demais subsistemas do satélite, e para isso, de maneira similar ao módulo desenvolvido para o FloripaSat-1, o OBDH 2 utiliza como sistema operacional o (FREERTOS... , s.d.). O sistema operacional de tempo real, nesse caso, é utilizado para garantir a execução das tarefas dentro dos limites de tempo estabelecidos, mesmo em caso de falhas (MARCELINO *et al.*, 2020).

O projeto e a implementação do OBDH 2 está disponível em seu repositório (SPACELAB, 2020b).

## 5.2 ANÁLISE ESTÁTICA

Como apresentado na Seção 4.7, a análise estática de código fornece um meio de obter informações sobre o código desenvolvido, encontrar e corrigir erros potenciais sem precisar compilar o código, o que se mostra bastante benéfico em projetos complexos e interdependentes, como *firmware* do FloripaSat-2. O uso de ferramentas de análise estática também garante maior precisão e segurança em códigos críticos. (WICHMANN *et al.*, 1995). Segundo os autores, a análise estática tem duas características principais que devem ser mantidas em mente quando utilizadas: "natureza" e "profundidade" (*nature* e *depth*, no original):

natureza: é o objetivo amplo da análise, que pode ser adequação a algum padrão de linguagem, ou a corretude de algum aspecto em relação à especificação do programa;

profundidade: indica a profundidade semântica da análise. Por exemplo, a análise de layout de um programa é rasa (...) enquanto provar um programa tem profundidade. (WICHMANN *et al.*, 1995).

Wichman *et. al.* relatam que análise estática é efetiva e complementar à testagem dinâmica, e seu uso é recomendado no desenvolvimento de sistemas críticos.

Como mencionado na Seção 4.7, os repositórios do FloripaSat-2 utilizam a ferramenta CPPCheck para análise estática de código, que verifica questões problemas com limites, tipagem, retornos e chamadas de funções. A lista completa de verificações feitas pode ser encontrada na página oficial da ferramenta, disponível em (CPPCHECK. . . , s.d.). Até o momento da escrita desta seção, a análise estática está implementada no módulo OBDH 2, onde é feita a verificação de todo o código implementado para o subsistema, com exceção dos testes. Seu uso foi configurado no modelo de automatização descrito na Seção 4.2 e detalhado na Seção 5.4.

## 5.3 TESTES DE UNIDADE

### 5.3.1 CMocka

Para a implementação dos testes de unidade nos módulos do FloripaSat-2, foi utilizada a ferramenta (CMOCKA. . . , s.d.), que implementa um framework de testes cuja única dependência é a biblioteca padrão C. Além disso, os testes escritos através da biblioteca são compilados em executáveis *stand-alone* e, o mais importante para o projeto, suportam o uso de *Mock Objects*, programas que simulam de maneira controlada o funcionamento de um sistema real. No caso do FloripaSat-2, os *mockups*

foram utilizados com o intuito de simular interações, dependências e chamadas I/O de programas externos ao alvo do teste.

Testes de unidade tem melhores precisão e eficiência quando empregados de maneira isolada e contida, para testar componentes e funcionalidades singulares. O componente sob teste deve ser analisado de maneira pura sem dependências e comunicações externas, pois isso acarretaria em introduzir código não testado para avaliar o funcionamento correto do componente desejado. Essa introdução pode trazer consigo *bugs* e defeitos inesperados, para os quais o teste não está preparado para receber, impedindo o propósito do teste.

De maneira simples, para testar um determinado componente *a*, que depende dos componentes *b* e *c*, a melhor maneira de fazê-lo seria encontrar uma maneira de fazer com que *a* possa contornar *b* e *c*, sem precisar que eles sejam invocados durante sua execução. Essa funcionalidade, a qual o prof. Raul Wazlawick nomeia *stub* em seu livro (WAZLAWICK, 2019), é oferecida pela biblioteca Cmocka através dos *Mock Objects* (Seção 5.3.1.1)

#### 5.3.1.1 Um *Mock Object* na CMocka

Como discutido anteriormente, a maior utilidade dos objetos *mock* para o FloripaSat-2 é possibilitar a simulação de funções e componentes externos ao alvo do teste. Os *Mock Objects* entram no projeto como forma de isolar o teste, de forma que qualquer erro ou defeito de implementação encontrado esteja localizado no próprio componente a ser testado.

Dando continuidade ao exemplo iniciado na Seção 5.3.1, ao escrever o teste para o componente *a*, podemos criar um objeto *mock* para os componentes *b* e *c*, de modo que sempre que *a* realizar alguma chamada à *b* ou *c*, teremos controle sobre o envio e o retorno das chamadas, garantindo que somente o código de *a* estará sendo executado, sem que bugs externos atrapalhem seu processo de teste.

A CMocka utiliza a funcionalidade `-wrap` do ligador. o manual do *GNU Linker* diz:

```
-wrap=symbol
```

Utilize uma função *wrapper* para *symbol*. Qualquer referência não definida à *symbol* será resolvida como `__wrap_symbol (...)`

(GNU..., 1994)

Segue então que, para testar de maneira isolada o componente *a* devemos implementar funções *mock* `__wrap_b` e `__wrap_c`. O teste de *a* seria compilado, então, através do comando

```
gcc a_test --wrap=b,--wrap=c
```

Dessa forma, *a* será compilado normalmente, mas chamadas à *b* e *c* serão resolvidas como chamadas aos seus respectivos *wrappers*.

Dentro de um arquivo de teste, essa interação é feita da seguinte maneira:

Uma função que testa um componente que fará uma chamada à um *wrapper* deve preparar anteriormente à execução do componente, o que a função *wrapper* receberá e/ou retornará. No teste isso é feito através das funções `expect_`. Consequentemente, a função *wrapper* deve verificar a existência e a corretude do parâmetro enviado, através da função `check_expected`. Essas duas funções permitem ao desenvolvedor e consequentemente ao próprio teste, saber de antemão qual o conteúdo das mensagens a ser trocadas, e qual o conteúdo das mesmas, podendo facilmente identificar se houve algum problema no percurso.

Caso a função *wrapper* deva retornar algum valor, isso pode ser feito dentro do teste através da função `will_return`, que especifica exatamente o que o *wrapper* retornará, e o que vai ser recebido pelo componente testado. O Código 5.1 dá continuidade ao exemplo anterior, exemplificando em código (relativamente) válido:

```
1      static void a_test (void **state) {
2          expect_value(__wrap_b, valor_b, 0);
3          will_return(__wrap_b, 1);
4
5          expect_value(__wrap_c, valor_c, 1);
6          will_return(__wrap_c, 0);
7
8          assert_return_code(a(0, 1), 0);
9      }
10
11     int __wrap_b(int valor_b) {
12         check_expected(valor_b)
13         return mock_type(int);
14     }
15
16     int __wrap_c(int valor_c) {
17         check_expected(valor_c);
18         return mock_type(int);
19     }
20
21     int a(int valor_b, int valor_c) {
22         int return_b = b(valor_b);
23         int return_c = b(valor_c);
24
25         return 0;
26     }
27
```

Código 5.1 – Exemplo de implementação e chamada de *mock objects*

Exemplos dessas funções implementadas em testes reais no módulo EPS 2 do FloripaSat-2 estão apresentados nos trechos de código 5.2 e 5.3.

```

1 int __wrap_adc_read(adc_port_t port, uint16_t *val)
2 {
3     check_expected(port);
4
5     uint16_t adc_val = mock_type(uint16_t);
6
7     if (val != NULL)
8     {
9         *val = adc_val;
10    }
11
12    return mock_type(int);
13 }
```

Código 5.2 – *Mock Function* do driver *adc*

Neste trecho de código do arquivo `adc_wrap.c` (disponível integralmente no apêndice A), É possível observar os mesmos princípios discutidos no exemplo anterior sendo aplicados na prática para testar os componentes. O *ADC* nesse caso se refere à um conversor analógico-digital (*analog-digital converter*).

No EPS, o *device* `temp_sensor.c` utiliza o driver *adc* para ler alguns sensores de temperatura, como parte da funcionalidade de controle térmico da bateria, conforme apresentado na Seção 5.1.1. Dessa forma, para testar o `temp_sensor`, foram implementadas funções *wrapper* para todas as chamadas ao driver *adc*.

```

1 static void temp_mcu_read_c_test(void **state)
2 {
3     uint16_t i = 0;
4
5     for (i = TEMP_SENSOR_ADC_MIN_VAL; i <= TEMP_SENSOR_ADC_MAX_VAL; ++i)
6     {
7         expect_value(__wrap_adc_read, port, TEMP_SENSOR_ADC_PORT);
8
9         will_return(__wrap_adc_read, i);
10        will_return(__wrap_adc_read, 0);
11        will_return(__wrap_adc_temp_get_mref, TEMP_SENSOR_ADC_MREF_VAL);
12        will_return(__wrap_adc_temp_get_nref, TEMP_SENSOR_ADC_NREF_VAL);
13
14        int16_t temp_c = INT16_MAX;
15
16        int result = temp_mcu_read_c(&temp_c);
17
18        assert_return_code(result, 0);
19        assert_in_range(temp_c + 300, TEMP_SENSOR_MIN_VAL_C + 300,
TEMP_SENSOR_MAX_VAL_C + 300);
```

```
20     }  
21 }
```

Código 5.3 – função teste temp\_sensor

## 5.4 AUTOMAÇÃO

Como descrito anteriormente, o objetivo de automatizar a execução dos testes é potencializar os ganhos de confiabilidade nos sistemas desenvolvidos, além de garantir a corretude do código produzido através de uma cobertura de testes elevada. Esses objetivos foram almejados utilizando a ferramenta GitHub Actions, aproveitando-se do fato de que o desenvolvimento da missão já estava anteriormente hospedada no GitHub.

### 5.4.1 GitHub Actions

Disponibilizada em 2019, GitHub Actions é uma API *event-driven* para automatizar *workflows* de desenvolvimento, que podem ser configurados para executar diversas atividades, a partir de diferentes tipos de eventos.

Um evento, como definido pelos autores Kinsman *et. al.*(2021):

É uma atividade específica que dispara a execução de um *workflow* (...) Por exemplo, um *workflow* é disparado quando um *pull-request* é criado, ou quando ocorre um *merge* na *branch* principal do repositório. (KINSMAN *et al.*, 2021)

Os *workflows* executados pela ferramenta são escritos em linguagem *YAML*, e são implementados no diretório `.github/workflows`, na raiz do repositório.

Um repositório pode ter um ou mais *workflows*. De fato, os repositórios do FloripaSat-II, especialmente o módulo OBDH 2 (SPACELAB, 2020b), contem três *workflows* distintos: um para execução dos testes de unidade dos *drivers*, o segundo com a mesma finalidade, para os testes de *devices*, e um terceiro para a execução da análise estática do código. Após a execução dos *workflows*, os resultados são apresentados em uma tela com *logs*. Existem outras formas de tratar as mensagens de execução, mas a missão optou por visualizar as mensagens desta maneira, que não envolve outras configurações. A figura Figura 4 demonstra um exemplo de mensagem de *log* após a execução com sucesso de um *workflow* de testes.

Os *workflows* nos repositórios da missão são organizados em duas etapas distintas, separadas em Pré Execução Seção 5.4.2 e Execução propriamente dita Seção 5.4.3.

Figura 4 – Tela de resultados após a execução de um *workflow*.

```

run-tests (antenna_test, antenna_unit_test, ./firmware/tests/devices/antenna_unit_test)
succeeded 7 hours ago in 40s

> ✓ Set up job
> ✓ Run actions/checkout@v2
> ✓ Install CMocka
> ✓ Signal make target
> ✓ Generate Test File
> ✓ Signal running test
> ✓ Run test
  1 ▶ Run $TEST_PATH
  8 [=====] Running 3 test(s).
  9 [ RUN      ] antenna_init_test
 10 [      OK   ] antenna_init_test
 11 [ PASSED   ] 3 test(s).
 12 [ RUN      ] antenna_get_data_test
 13 [      OK   ] antenna_get_data_test
 14 [ RUN      ] antenna_deploy_test
 15 [      OK   ] antenna_deploy_test
 16 [=====] 3 test(s) run.
> ✓ Post Run actions/checkout@v2
> ✓ Complete job

```

Fonte: OBDH 2 (SPACELAB, 2020b)

#### 5.4.2 Pré Execução

Na etapa de Pré Execução, são realizadas algumas tarefas para preparar o *workflow*. Configurações iniciais, como escolha de sistema operacional e download e instalação de pacotes necessários foram omitidas, por se tratarem de configuração padrão da ferramenta, e não tem relevância para a execução do modelo desenvolvido.

A primeira tarefa é a elaboração de uma *Job Matrix*, que se trata de uma estrutura de dados empregada pela ferramenta para estabelecer um conjunto de *jobs*, como são chamadas as ações da ferramenta. Um exemplo *job*, no contexto do FloripaSat-II, seria um teste a ser executado. A *Job Matrix* é utilizada, então, para dividir o *workflow* de modo que cada arquivo de teste seja encapsulado em uma sub-tarefa distinta, que são executadas de maneira paralela. Optou-se por esse modelo como maneira de garantir execução e análise eficientes de e de maneira independente. A Tabela 1 e a Tabela 2 apresentam uma comparação entre as execuções do workflow no repositório do OBDH 2.0 atual, e uma versão sequencial implementada especialmente para este



teste.

Modo de Execução	Min	Max	Média
Sequencial	38	69	50.27
Paralela (Job Matrix)	74	434 <sup>1</sup>	128

Tabela 1 – Tabela comparativa de tempos de execução em segundos do workflow de testes unitários dos *drivers* do OBDH 2.0

Modo de Execução	Min	Max	Média
Sequencial	27	79	47.2
Paralela (Job Matrix)	57	105	72.8

Tabela 2 – Tabela comparativa de tempos de execução em segundos do workflow de testes unitários dos *devices* do OBDH 2.0

O *workflow* que implementa a execução sequencial dos testes se mostrou comparativamente mais rápido à execução paralela. A razão da escolha do uso da *Job Matrix* foi a possibilidade de granularizar a execução, controlando de maneira individual se um teste deve ou não ser executado, além de ser possível analisar os resultados da execução de maneira individual. Como cada item da matriz é visualizada como um *workflow* separado nos logs do GitHub, se torna trivial encontrar o teste desejado e analisar o resultado de sua execução. Na forma sequencial, por exemplo, a execução se dá em um único *workflow*, e os testes e resultados devem ser buscados no log.

Nesse contexto, um *job* principal faz o levantamento de todos os arquivos de teste disponíveis para serem executados, e para cada arquivo é gerada uma entrada na *job matrix*, de modo que cada teste seja compilado e executado em um *job* isolado.

O trecho de código Código 5.4 representa a seção do *workflow* responsável por gerar o conteúdo da *Job Matrix*, que no caso é populada com informações sobre os testes a serem executados.

```

1  # Generates Matrix
2  # This job executes the 'deployJSON.py' script
3  # which compiles a list of all files ending in '_test.c'
4  # in a given directory and includes them in a .json file
5  # along with the path to the executable file.
6  generate-matrix:
7    name:
8
9    runs-on: ubuntu-latest

```

<sup>1</sup> O tempo de duração de 434 segundos foi possivelmente uma anomalia decorrente de alguma instabilidade na plataforma, visto que nenhuma das outras 10 execuções desse mesmo módulo apresentou valores próximos a este. Para fins de comparação, foi realizada uma nova medição desse mesmo workflow, e a média recalculada descartando o valor anômalo foi de 97.4 segundos. A lista completa das execuções pode ser encontrada nas tabelas 3 e 4 no apêndice D

```

10
11     outputs:
12         matrix: ${ steps.set-matrix.outputs.matrix }
13
14     steps:
15         # Checks-out the repository under $GITHUB_WORKSPACE, so the
16         ↪ job can navigate it
17         - uses: actions/checkout@v2
18
19         - name: Create JSON file
20           run: python3 .github/workflows/deployJSON.py --source
21           ↪ firmware/tests/devices/
22
23         - name: Resulting JSON file for matrix generation
24           run: echo "$(cat .github/workflows/test-list.json)"
25
26         # Set the matrix output from the JSON (manipulated to remove
27         ↪ spaces and replace \n -> %0A, " -> \")
28         - id: set-matrix
29           name: Set matrix output from the JSON file
30           run: echo "::set-output name=matrix::$( echo "$(cat
31           ↪ .github/workflows/test-list.json)" | sed
32           ↪ ':a;N;$!ba;s/\n/%0A/g' )"

```

Código 5.4 – Geração da Job Matrix no workflow de autmação dos testes de *devices* do OBDH

A execução desta etapa, que por sua vez invoca um script Python que lê o diretório onde o teste está implementado, cuja localização no repositório é informada no momento da execução. Assim, cada *workflow* informa o caminho desejado. Por exemplo, no caso do OBDH 2.0, o workflow de *drivers* informaria o caminho `firmware/tests/drivers`, enquanto o workflow de *devices* deverá informar `firmware/tests/devices`. Essa configuração deve ser feita de forma manual na implementação dos *workflows* individuais, e segue o padrão estabelecido e descrito na Seção 4.4.

A execução deste script resultará em uma lista no formato *.json* com os seguintes componentes:

- *name*: o nome do arquivo de teste fonte.c, *sem a extensão*;
- *test\_name*: o nome do executável após a compilação do arquivo fonte;
- *path*: o diretório onde se encontra o executável;

Um exemplo desta lista gerada pelo workflow de *devices* do OBDH 2.0 está disponível no trecho de código Código 5.5.

```
1  "include": [  
2    {  
3      "name": "antenna_test",  
4      "test_name": "antenna_unit_test",  
5      "path": "./firmware/tests/devices/antenna_unit_test"  
6    },  
7    {  
8      "name": "current_sensor_test",  
9      "test_name": "current_sensor_unit_test",  
10     "path":  
11     ↪ "./firmware/tests/devices/current_sensor_unit_test"  
12   },  
13   {  
14     "name": "eps_test",  
15     "test_name": "eps_unit_test",  
16     "path": "./firmware/tests/devices/eps_unit_test"  
17   },  
18 ]
```

Código 5.5 – Lista de testes gerada pelo script de identificação de arquivos

O arquivo *.json* apresenta o resultado da execução do script, onde cada elemento de *include* é um dicionário contendo o nome do arquivo fonte, o nome e a localização do executável.

Esse arquivo é lido nos próximos passos do workflow para gerar os *jobs* individuais com cada um desses dicionários, dessa forma cada *job* lida apenas com um executável.

### 5.4.3 Execução

Após as configurações realizadas na etapa de Pré Execução, o *workflow* realiza a compilação e a execução dos arquivos listados anteriormente, localizados no arquivo *.json* gerado.

Para efeitos de ilustração, assumiremos um commit na branch *dev\_firmware* do repositório do OBDH 2 como evento *e*, que aciona a execução de um *workflow w*.

Após a etapa de Pré Execução, todos os arquivos de teste dos *drivers* e *devices* do OBDH2 estarão listados no arquivo *.json*, e a *Job Matrix* estará populada com os nomes e os caminhos de cada um deles. Cada entrada na matriz é, então, dividida em uma sub tarefa.

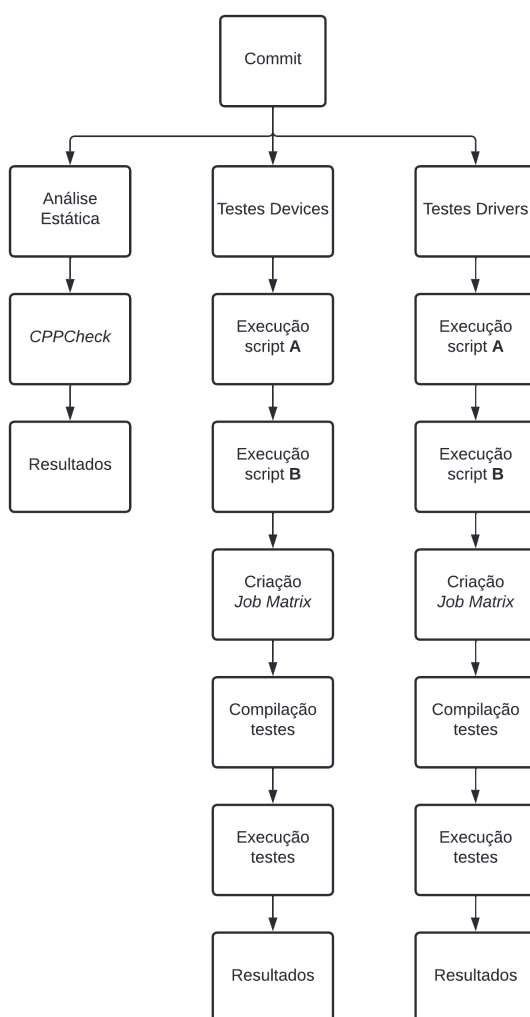
Como cada tarefa é encarada como um *workflow* distinto sob o ponto de vista de execução, todas elas precisam realizar algumas configurações iniciais como as

descritas anteriormente, de download e instalação de bibliotecas. Após isso, o teste é compilado, e o binário gerado é executado. Ao final da execução, um registro com informações sobre o *workflow* é gerado (Figura 4).

Caso algum dos testes apresente falha, o *workflow* registrará um erro de execução, mas isso ocorrerá apenas após a finalização de todos os testes.

A Figura 5 ilustra um fluxo de execuções seguindo as etapas discutidas nas últimas seções.

Figura 5 – Fluxo de execução do modelo de automação



Fonte: O autor

## **6 RESULTADOS**

### **6.1 EXECUÇÕES**

Os workflows no repositório do obdh (onde os testes já estão mais consolidados) foram executadas 447 vezes até a data da escrita deste trabalho, sendo que 44 execuções representaram falhas. O que representa cerca de 10% das execuções.

No EPS, 77 execuções com 15 falhas, uma taxa de aproximadamente 20%. O Número de execuções é menor, e a taxa de falhas é maior, porém o eps2 está em desenvolvimento ativo, com os testes ainda sendo escritos e implementados.

### **6.2 BUGS ENCONTRADOS**

### **6.3 CONSIDERAÇÕES**

Os workflows foram amplamente utilizados durante os ciclos de desenvolvimento do FloripaSat-2 e serviram ativamente para encontrar e solucionar problemas na implementação.

## 7 CONSIDERAÇÕES FINAIS

Os trabalhos relacionados escolhidos Capítulo 3 apresentam um panorama inicial bastante informativo, quando analisados em conjunto:

- A relevância de missões CubeSat cresce a cada ano, observada pelo crescente volume de lançamentos;
- São projetos com potencial educacional elevado, sendo usados por instituições de ensino para capacitar e treinar estudantes e também desenvolver, testar novas tecnologias e processos de desenvolvimento.
- Se tratam de sistemas críticos, cuja confiabilidade é fator essencial para o sucesso da missão

Os artigos elaborados pela equipe do FloripaSat-I demonstram a enorme quantidade de conhecimento gerada, bem como a experiência adquirida em projeto e desenvolvimento de tecnologias espaciais. Os resultados preliminares obtidos após o lançamento do FloripaSat-I também se mostraram essenciais, ao revelar pontos de atenção e dificuldades a serem melhor tratados na missão atual.

Percebeu-se também que, embora exista uma biblioteca ampla sobre testes de hardware de *CubeSats*, a quantidade de material distinto que foque apenas (ou principalmente) em teste de software espacial é comparativamente menor. O material encontrado, no entanto, revela os benefícios e melhorias que o modelo proposto por esse trabalho potencialmente proporciona. Espera-se que esse trabalho seja entendido como uma continuidade desse pensamento, no estudo realizado nos testes de unidade do FloripaSat-II.

O estudo destes trabalhos relacionados evidência também a importância de um projeto de testes estruturado. Mais de uma vez, nos artigos selecionados anteriormente foi descrito que a etapa de testes foi importante para a descoberta e correção de erros de design, sejam eles de hardware ou software. É nessa vertente que este trabalho propôs e implementou uma nova aproximação para os testes unitários do FloripaSat-2, por meio de um sistema de workflows que permite a execução de testes de maneira automática.

É possível verificar a importância da execução destes *workflows*, que permitem a centralização e a transparência dos testes escritos, de uma forma que seja facilmente identificável quando um erro de implementação ocorra, e igualmente fácil a todos os desenvolvedores analisarem e terem a oportunidade de contribuir com a identificação e correção dos erros.

Além disso, publicações e entregas de código podem ser agilizadas, sendo que uma vez que os testes já estejam escritos e implementados no repositório, os com-

mits podem ser feitos de forma experimental, e o *workflow* apresentaria um relatório, apontando se e quais falhas foram implementadas.

## 7.1 TRABALHOS FUTUROS

Como trabalhos futuros, sugere-se investigar as possibilidades de expansão do modelo, como descrito nas seções a seguir:

### 7.1.1 Testes de Integração

Apesar deste trabalho manter foco em testes de *software*, recomenda-se explorar possibilidades de utilizar a ferramenta GitHub Actions para automatizar testes de integração, gravando e executando o código no *hardware* do satélite. Isso deve ser feito estabelecendo uma máquina física do laboratório como servidor do *workflow*, já que ele deve ter acesso físico ao *hardware*. Este trabalho permitiria verificar a compatibilidade do *firmware* como *hardware*, investigando possíveis erros de compilação e interação entre os subsistemas.

### 7.1.2 FlatSat

Uma das etapas descritas no plano de desenvolvimento do FloripaSat-II é o teste no modelo FlatSat, onde os subsistemas do satélite são conectados para simular a execução e interação final entre os módulos. Uma possível ideia de expansão do projeto é desenvolver um *workflow* para monitorar e analisar esses testes de maneira remota, sem que seja necessária dedicação constante à montagem, que estará sendo lidada pela ferramenta de automação.

### 7.1.3 Containerização

Containers são similares às máquinas virtuais, com a vantagem de não consumirem tanto tempo e recursos como VMs convencionais, enquanto seguem os mesmos princípios de virtualização. Em especial, containers oferecem a capacidade de serem interconectados (PAHL, 2015). Sugere-se como trabalho futuro explorar a containerização dos *workflows* propostos por esse trabalho, de modo a fazer uso das ferramentas de computação em nuvem e *Platform as a Service* como forma de automatizar os testes do FloripaSat-II.

## REFERÊNCIAS

"MARCELINO, Gabriel Mariano; MORSCH, Filho Edemar; SARA, Vega-Martinez; PIO, De Mattos André Martins; ORIEL, Seman Laio; KESSLER, Slongo Leonardo; AUGUSTO, Bezerra Eduardo". "Qualification and validation test methodology of the open-source CubeSat FloripaSat-I". **"Journal of Systems Engineering and Electronics"**, v. 31, n. 6, p. 1230–1244, 2020. DOI: 10.23919/JSEE.2020.000103. Disponível em: <https://ieeexplore.ieee.org/document/9316404>.

ALANAZI, Abdulaziz; STRAUB, Jeremy. Engineering Methodology for Student-Driven CubeSats. **Aerospace**, v. 6, n. 5, 2019. ISSN 2226-4310. DOI: 10.3390/aerospace6050054. Disponível em: <https://www.mdpi.com/2226-4310/6/5/54>.

AYEWAH, Nathaniel; PUGH, William; HOVEMEYER, David; MORGENTHALER, J. David; PENIX, John. Using Static Analysis to Find Bugs. **IEEE Software**, v. 25, n. 5, p. 22–29, 2008. DOI: 10.1109/MS.2008.130.

BURT, Robert. **Distributed Electrical power systems in cubesat applications**. 2011. Diss. (Mestrado) – Utah State University.

CANADIAN SPACE AGENCY - CSA. **What is a CubeSat**. [S.l.: s.n.], 2018. [asc-csa.gc.ca/eng/satellites/cubesat/what-is-a-cubesat.asp](http://asc-csa.gc.ca/eng/satellites/cubesat/what-is-a-cubesat.asp). Acessado em 30/08/2021.

CHEN, M.-H.; LYU, M.R.; WONG, W.E. Effect of code coverage on software reliability measurement. **IEEE Transactions on Reliability**, v. 50, n. 2, p. 165–170, 2001. DOI: 10.1109/24.963124.

CMOCKA. [S.l.: s.n.]. <https://api.cmocka.org/>. Acessado em 14/06/2022.

CPPCHECK - A tool for static C/C++ code analysis. [S.l.: s.n.]. <http://cppcheck.net/>. Acessado em 29/06/2022.

FEWSTER, Mark; GRAHAM, Dorothy. **Software test automation**. [S.l.]: Addison-Wesley Reading, 1999.

FREERTOS. [S.l.: s.n.]. <https://freertos.org/index.html>. Acessado em 21/06/2022.



GITHUB. **GitHub Actions**. [S.l.: s.n.]. [github.com/features/actions](https://github.com/features/actions). Acessado em 01/09/2021.

GNU Linker. [S.l.: s.n.], 1994.

[https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html\\_mono/ld.html](https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html). Acessado em 01/07/2022.

GONZALEZ, Carlos E.; ROJAS, Camilo J.; BERGEL, Alexandre; DIAZ, Marcos A. An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites. **IEEE Access**, v. 7, p. 126409–126429, 2019. DOI: 10.1109/ACCESS.2019.2927931.

ISTSAT-1. [S.l.: s.n.], 2021.

<https://istsat-one.tecnico.ulisboa.pt/~istsat-one.daemon/>. Acessado em 28/02/2022.

JOHNSTONE, Alicia. **CubeSat Design Specification (1U – 12U) REV 14 CP-CDS-R14**. San Luis Obispo, CA, EUA, 2020.

JPL. **Mars Cube One (MarCO)**. [S.l.: s.n.], 2018.

<https://www.jpl.nasa.gov/missions/mars-cube-one-marco>. Acessado em 21/02/2022.

KINSMAN, Timothy; WESSEL, Mairieli; GEROSA, Marco A.; TREUDE, Christoph. How Do Software Developers Use GitHub Actions to Automate Their Workflows? *In*: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). [S.l.: s.n.], 2021. P. 420–431. DOI: 10.1109/MSR52588.2021.00054.

KULU, ERIK. **Nanosats Database**. [S.l.: s.n.], 2021. <https://www.nanosats.eu/>. Acessado em 08/09/2021.

MALAIYA, Y.K.; LI, M.N.; BIEMAN, J.M.; KARCICH, R. Software reliability growth with test coverage. **IEEE Transactions on Reliability**, v. 51, n. 4, p. 420–426, 2002. DOI: 10.1109/TR.2002.804489.

MARCELINO, Gabriel Mariano; MORSCH FILHO, Edemar; MARTINEZ, Sara Vega; SEMAN, Laio Oriel; BEZERRA, Eduardo Augusto. In-orbit preliminary results from the open-source educational nanosatellite FloripaSat-I. **Acta Astronautica**, v. 188, p. 64–80, 2021. ISSN 0094-5765. DOI:

<https://doi.org/10.1016/j.actaastro.2021.07.001>. Disponível em:  
<https://www.sciencedirect.com/science/article/pii/S0094576521003489>.

MARCELINO, Gabriel Mariano; VEGA-MARTINEZ, Sara; SEMAN, Laio Oriel; KESSLER SLONGO, Leonardo; BEZERRA, Eduardo Augusto. "A Critical Embedded System Challenge: The FloripaSat-1 Mission". **"IEEE Latin America Transactions"**, v. 18, n. 02, p. 249–256, 2020. DOI: 10.1109/TLA.2020.9085277. Disponível em: <https://ieeexplore.ieee.org/document/9085277>.

MONTEIRO, João P.; ROCHA, Rui M.; SILVA, Alexandre; AFONSO, Rúben; RAMOS, Nuno. Integration and Verification Approach of ISTSat-1 CubeSat. **Aerospace**, v. 6, n. 12, 2019. ISSN 2226-4310. DOI: 10.3390/aerospace6120131. Disponível em: <https://www.mdpi.com/2226-4310/6/12/131>.

PAHL, Claus. Containerization and the PaaS Cloud. **IEEE Cloud Computing**, v. 2, n. 3, p. 24–31, 2015. DOI: 10.1109/MCC.2015.51.

PANGA, WJ; BO, B; MENG, X; YU, XZ; GUO, J; ZHOU, J. Boom of the CubeSat: A Statistic Survey of CubeSats Launch in 2003–2015. *In*: PROCEEDINGS of the 67th International Astronautical Congress (IAC), Guadalajara, Mexico. [S.l.: s.n.], 2016. P. 26–30.

SPACELAB. **Electrical Power System 2.0**. [S.l.: s.n.], 2020a.  
<https://github.com/spacelab-ufsc/eps2>. Acessado em 11/03/2022.

SPACELAB. **On-Board Data Handling 2.0**. [S.l.: s.n.], 2020b.  
<https://github.com/spacelab-ufsc/obdh2>. Acessado em 11/03/2022.

SPACELAB - UFSC. **FloripaSat-2 Doc**. [S.l.: s.n.], 2021.  
<https://spacelab-ufsc.github.io/floripasat2-doc/slb-fsat2-doc-v0.2.pdf>.  
Acessado em 30/08/2021.

TSCHERBAKOVA, N. N.; BELETSKII, V. V.; SOZONOV, V. V. Stabilization of Heliosynchronous Orbits of an Earth's Artificial Satellite by Solar Pressure. **Cosmic Research**, v. 17, n. 4, p. 393–401, 1998. DOI: 10.1109/MS.2008.130.

WAZLAWICK, Raul Sidnei. **Engenharia de Software: Conceitos e Práticas**. 2. ed. [S.l.]: Elsevier, 2019. ISBN 978-8535260847.

WICHMANN, B.; CANNING, A.A.; CLUTTERBUCK, D.L.; WINSBORROW, L.A.;  
WARD, N.J.; MARSH, William. Industrial perspective on static analysis. **Software  
Engineering Journal**, v. 10, p. 69–75, abr. 1995. DOI: 10.1049/sej.1995.0010.

# **Apêndices**

## APÊNDICE A – WORKFLOW DE TESTES DE UNIDADE PARA O MÓDULO OBDH 2.0

Este capítulo apresenta os códigos necessários para a execução do *workflow* de automação de testes de unidade do módulo OBDH 2.0 (SPACELAB, 2020b) do FloripSat-2

### A.1 UNIT-TESTS-DEVICES.YML

---

```

1  #
2  # unit-tests-devices.yml
3  #
4  # Copyright (C) 2021, SpaceLab.
5  #
6  # This file is part of OBDH 2.0.
7  #
8  # OBDH 2.0 is free software: you can redistribute it and/or modify
9  # it under the terms of the GNU General Public License as published by
10 # the Free Software Foundation, either version 3 of the License, or
11 # (at your option) any later version.
12 #
13 # OBDH 2.0 is distributed in the hope that it will be useful,
14 # but WITHOUT ANY WARRANTY; without even the implied warranty of
15 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
16 # GNU General Public License for more details.
17 #
18 # You should have received a copy of the GNU General Public License
19 # along with OBDH 2.0. If not, see <http://www.gnu.org/licenses/>.
20 #
21 #
22
23
24 name: Devices Unit Tests
25
26 on:
27   push:
28     branches: [ dev_firmware ]
29   pull_request:
30     branches: [ master, dev, dev_firmware]
31
32   # 'workflow_dispatch' allows manual execution
33   # of this workflow under the repository's 'Actions' tab
34   workflow_dispatch:
35
36 jobs:
37
38   # Generates Matrix
```

```

39  # This job executes the 'deployJSON.py' script
40  # which compiles a list of all files ending in '_test.c'
41  # in a given directory and includes them in a .json file
42  # along with the path to the executable file.
43  generate-matrix:
44      name:
45
46      runs-on: ubuntu-latest
47
48      outputs:
49          matrix: ${{ steps.set-matrix.outputs.matrix }}
50
51      steps:
52          # Checks-out the repository under $GITHUB_WORKSPACE, so the job can navigate it
53          - uses: actions/checkout@v2
54
55          - name: Create JSON file
56            run: python3 .github/workflows/deployJSON.py --source firmware/tests/devices/
57
58          - name: Resulting JSON file for matrix generation
59            run: echo "$(cat .github/workflows/test-list.json)"
60
61          # Set the matrix output from the JSON (manipulated to remove spaces and replace
62          ↪ \n -> %0A, " -> \")
63          - id: set-matrix
64            name: Set matrix output from the JSON file
65            run: echo "::set-output name=matrix::$( echo "$(cat
66            ↪ .github/workflows/test-list.json)" | sed ':a;N;$!ba;s/\n/%0A/g' )"
67
68          # This job reads the matrix containing the paths
69          # created by the previous job and runs each program
70          # individually, i.e. spawning one job for every
71          # test file included in the matrix.
72      run-tests:
73          name: run-tests
74          needs: generate-matrix
75          runs-on: ubuntu-latest
76
77          strategy:
78              fail-fast: false
79
80          matrix: ${{fromJson(needs.generate-matrix.outputs.matrix)}}
81
82      env:
83          MAKE_TARGET: ${{ matrix.name }}
84          TEST_FILE: ${{ matrix.test_name }}
85          TEST_PATH: ${{ matrix.path }}

```

---

```
86     steps:
87       - uses: actions/checkout@v2
88         # Install required libs
89       - name: Install CMocka
90         run: sudo apt-get install libcmocka0 libcmocka-dev
91
92       - name: Signal make target
93         run: echo "Generating make file for $MAKE_TARGET"
94
95       - name: Generate Test File
96         run: cd firmware/tests/devices && make $MAKE_TARGET
97
98       - name: Signal running test
99         run: echo "Running $TEST_FILE"
100
101       - name: Run test
102         run: $TEST_PATH
```

---

## APÊNDICE B – DEPLOY JSON

### B.1 DEPLOYJSON.PY

---

```

1      #
2      # deployJSON.py
3      #
4      # Copyright (C) 2021, SpaceLab.
5      #
6      # This file is part of OBDH 2.0.
7      #
8      # OBDH 2.0 is free software: you can redistribute it and/or modify
9      # it under the terms of the GNU General Public License as published by
10     # the Free Software Foundation, either version 3 of the License, or
11     # (at your option) any later version.
12     #
13     # OBDH 2.0 is distributed in the hope that it will be useful,
14     # but WITHOUT ANY WARRANTY; without even the implied warranty of
15     # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
16     # GNU General Public License for more details.
17     #
18     # You should have received a copy of the GNU General Public License
19     # along with OBDH 2.0. If not, see <http://www.gnu.org/licenses/>.
20     #
21     #
22
23     import sys
24     import os
25     import json
26
27     if len(sys.argv) <= 2:
28         print("\nWrong arguments")
29         print("Use: python3 test-deployer --source <target directory>\n")
30
31         sys.exit(1)
32     else:
33         if sys.argv[1] == '--source':
34             # Directory path and list
35             path = sys.argv[2]
36             if path.startswith('/'):
37                 path = '.' + path
38             elif not path.startswith('./'):
39                 path = './' + path
40             file_dir = os.listdir(path)
41
42             files = {
43                 "include": []
44             }

```



```
45
46     # Naming convention:
47     #   - test files should end with '_test.c';
48     #   - and executables should be named '<target>_unit_test
49     #   - makefile commands should be 'make <target>' for each test file
50     #
51     # This script should work for all folders that follow these guidelines
52     # otherwise this script will have to be edited to function properly
53
54     for file in file_dir:
55         if file.endswith('_test.c'):
56             file_name = file.replace('.c', '')
57             test_file = file_name.replace('_test', '_unit_test')
58             file_path = path + test_file
59
60             file_info = {
61                 "name": file_name,
62                 "test_name": test_file,
63                 "path": file_path
64             }
65             files['include'].append(file_info)
66
67     # Convert the dictionary to JSON format
68     json_target = json.dumps(files)
69
70     with open(".github/workflows/test-list.json", "w") as json_file:
71         json_file.write(json_target)
72     json_file.close()
73
74     print("JSON created successfully for folder " + sys.argv[2])
```

---

## APÊNDICE C – ADS1248\_WRAP.C

---

```

1  /*
2   * ads1248_wrap.c
3   *
4   * Copyright (C) 2021, SpaceLab.
5   *
6   * This file is part of EPS 2.0.
7   *
8   * EPS 2.0 is free software: you can redistribute it and/or modify
9   * it under the terms of the GNU General Public License as published by
10  * the Free Software Foundation, either version 3 of the License, or
11  * (at your option) any later version.
12  *
13  * EPS 2.0 is distributed in the hope that it will be useful,
14  * but WITHOUT ANY WARRANTY; without even the implied warranty of
15  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
16  * GNU General Public License for more details.
17  *
18  * You should have received a copy of the GNU General Public License
19  * along with EPS 2.0. If not, see <http://www.gnu.org/licenses/>.
20  *
21  */
22
23 /**
24  * \brief ADS1248 driver wrap implementation.
25  *
26  * \author Lucas Zacchi de Medeiros <lucas.zacchi@spacelab.ufsc.br>
27  *
28  * \version 0.1.0
29  *
30  * \date 2021/09/06
31  *
32  * \defgroup ads1248_wrap ADS1248 Wrap
33  * \ingroup tests
34  * \{
35  */
36
37
38 #include <stdarg.h>
39 #include <stddef.h>
40 #include <stdint.h>
41 #include <setjmp.h>
42 #include <float.h>
43 #include <cmocka.h>
44
45
46 #include "ads1248_wrap.h"

```

```
47 int __wrap_ads1248_init(ads1248_config_t *config)
48 {
49     return mock_type(int);
50 }
51
52 int __wrap_ads1248_reset(ads1248_config_t *config, ads1248_reset_mode_t mode)
53 {
54     check_expected_ptr(mode);
55
56     return mock_type(int);
57 }
58
59 int __wrap_ads1248_config_regs(ads1248_config_t *config)
60 {
61     return mock_type(int);
62 }
63
64 int __wrap_ads1248_read_regs(ads1248_config_t *config, uint8_t *rd)
65 {
66     check_expected(rd);
67     return mock_type(int);
68 }
69
70 int __wrap_ads1248_read_data(ads1248_config_t *config, uint8_t *rd, uint8_t
    ↪ positive_channel)
71 {
72     check_expected(rd);
73     check_expected(positive_channel);
74
75     return mock_type(int);
76 }
77
78 int __wrap_ads1248_write_cmd(ads1248_config_t *config, ads1248_cmd_t cmd, uint8_t *rd,
    ↪ uint8_t positive_channel)
79 {
80     check_expected(cmd);
81     check_expected(positive_channel);
82
83     if (config != NULL)
84     {
85         if (cmd == ADS1248_CMD_RDATA)
86         {
87             config = 0;
88         }
89     }
90     return mock_type(int);
91 }
92
```

---

```
93  int __wrap_ads1248_set_powerdown_mode(ads1248_config_t *config, ads1248_power_down_t
    ↪ mode)
94  {
95      if (config != NULL)
96      {
97          config->spi_port = mock_type(gpio_pin_t);
98          config->start_pin = mock_type(gpio_pin_t);
99      }
100     check_expected(mode);
101     return mock_type(int);
102 }
103
104 /** \} End of ads1248_wrap group */
```

---

## APÊNDICE D – EXECUÇÕES DOS DRIVERS WORKFLOWS DO OBDH2

Nas tabelas 3 e 4 estão listadas as execuções utilizadas como parâmetro para os dados apresentados nas tabelas 1 e 2, respectivamente.

<b>Execuções</b>	<b>Sequencial</b>	<b>Paralela (<i>Job Matrix</i>)</b>
1	44	92
2	42	86
3	38	83
4	42	95
5	69	128
6	43	103
7	57	115
8	60	434
9	56	138
10	50	74
11	52	60

Tabela 3 – Execuções do workflow drivers

<b>Execuções</b>	<b>Sequencial</b>	<b>Paralela (<i>Job Matrix</i>)</b>
1	53	87
2	66	57
3	59	66
4	41	71
5	79	65
6	73	96
7	53	105
8	38	93
9	42	89
10	27	59

Tabela 4 – Execuções do workflow

# **Anexos**

## ANEXO A – DEVICE ADC

teste

```

1 /*
2  * adc.c
3  *
4  * Copyright (C) 2021, SpaceLab.
5  *
6  * This file is part of EPS 2.0.
7  *
8  * EPS 2.0 is free software: you can redistribute it and/or modify
9  * it under the terms of the GNU General Public License as published by
10 * the Free Software Foundation, either version 3 of the License, or
11 * (at your option) any later version.
12 *
13 * EPS 2.0 is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
16 * GNU General Public License for more details.
17 *
18 * You should have received a copy of the GNU General Public License
19 * along with EPS 2.0. If not, see <http://www.gnu.org/licenses/>.
20 *
21 */
22
23 /**
24  * \brief ADC driver implementation.
25  *
26  * \author Gabriel Mariano Marcelino <gabriel.mm8@gmail.com>
27  *
28  * \version 0.2.20
29  *
30  * \date 2021/06/11
31  *
32  * \addtogroup adc
33  * \{
34  */
35
36 #include <stdbool.h>
37
38 #include <hal/gpio.h>
39 #include <hal/adc10_a.h>
40 #include <hal/adc12_a.h>
41 #include <hal/ref.h>
42 #include <hal/tlv.h>
43
44 #include <config/config.h>

```

```

45 #include <system/sys_log/sys_log.h>
46
47 #include "adc.h"
48
49 bool adc_is_ready = false;
50
51 float adc_mref = 0;
52 float adc_nref = 0;
53
54 uint8_t adc_cal_bytes;
55 struct s_TLV_ADC_Cal_Data *adc_cal_data;
56
57 int adc_init(adc_port_t port, adc_config_t config)
58 {
59     if (adc_is_ready)
60     {
61         #if CONFIG_DRIVERS_DEBUG_ENABLED == 1
62             sys_log_print_event_from_module(SYS_LOG_WARNING, ADC_MODULE_NAME
63             , "ADC driver already initialized!");
64             sys_log_new_line();
65         #endif /* CONFIG_DRIVERS_DEBUG_ENABLED */
66         return 0;
67     }
68
69     /* Set port 6 and 7 pins as inputs */
70     GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P6, GPIO_PIN0 |
71     GPIO_PIN1 | GPIO_PIN2 | GPIO_PIN3 | GPIO_PIN4 | GPIO_PIN5 |
72     GPIO_PIN6 | GPIO_PIN7);
73     GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P7, GPIO_PIN4 |
74     GPIO_PIN5 | GPIO_PIN6 | GPIO_PIN7);
75
76     ADC12_A_init(ADC12_A_BASE, ADC12_A_SAMPLEHOLDSOURCE_SC,
77     ADC12_A_CLOCKSOURCE_ADC12OSC, ADC12_A_CLOCKDIVIDER_1);
78
79     ADC12_A_enable(ADC12_A_BASE);
80
81     ADC12_A_setupSamplingTimer(ADC12_A_BASE,
82     ADC12_A_CYCLEHOLD_768_CYCLES, ADC12_A_CYCLEHOLD_4_CYCLES,
83     ADC12_A_MULTIPLESAMPLESDISABLE);
84
85     ADC12_A_configureMemoryParam param = {0};
86
87     /* Temperature sensor */
88     param.memoryBufferControlIndex = ADC12_A_MEMORY_0;
89     param.inputSourceSelect = ADC12_A_INPUT_TEMPSENSOR;
90     param.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_INT;
91     param.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;

```



```
85     param.endOfSequence                = ADC12_A_NOTENDOFSEQUENCE;
86     ADC12_A_configureMemory(ADC12_A_BASE, &param);
87
88     /* Solar Panel -Y current sensor */
89     param.memoryBufferControlIndex      = ADC12_A_MEMORY_1;
90     param.inputSourceSelect             = ADC12_A_INPUT_A0;
91     param.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_EXT;
92     param.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
93     param.endOfSequence                = ADC12_A_NOTENDOFSEQUENCE;
94     ADC12_A_configureMemory(ADC12_A_BASE, &param);
95
96     /* Solar Panel +X current sensor */
97     param.memoryBufferControlIndex      = ADC12_A_MEMORY_2;
98     param.inputSourceSelect             = ADC12_A_INPUT_A1;
99     param.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_EXT;
100    param.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
101    param.endOfSequence                = ADC12_A_NOTENDOFSEQUENCE;
102    ADC12_A_configureMemory(ADC12_A_BASE, &param);
103
104    /* Solar Panel -X current sensor */
105    param.memoryBufferControlIndex      = ADC12_A_MEMORY_3;
106    param.inputSourceSelect             = ADC12_A_INPUT_A2;
107    param.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_EXT;
108    param.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
109    param.endOfSequence                = ADC12_A_NOTENDOFSEQUENCE;
110    ADC12_A_configureMemory(ADC12_A_BASE, &param);
111
112    /* Solar Panel +Z current sensor */
113    param.memoryBufferControlIndex      = ADC12_A_MEMORY_4;
114    param.inputSourceSelect             = ADC12_A_INPUT_A3;
115    param.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_EXT;
116    param.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
117    param.endOfSequence                = ADC12_A_NOTENDOFSEQUENCE;
118    ADC12_A_configureMemory(ADC12_A_BASE, &param);
119
120    /* Solar Panel -Z current sensor */
121    param.memoryBufferControlIndex      = ADC12_A_MEMORY_5;
122    param.inputSourceSelect             = ADC12_A_INPUT_A4;
123    param.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_EXT;
124    param.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
125    param.endOfSequence                = ADC12_A_NOTENDOFSEQUENCE;
126    ADC12_A_configureMemory(ADC12_A_BASE, &param);
127
128    /* Solar Panel +Y current sensor */
129    param.memoryBufferControlIndex      = ADC12_A_MEMORY_6;
130    param.inputSourceSelect             = ADC12_A_INPUT_A5;
131    param.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_EXT;
```

```

132     param.negativeRefVoltageSourceSelect    = ADC12_A_VREFNEG_AVSS;
133     param.endOfSequence                     = ADC12_A_NOTENDOFSEQUENCE;
134     ADC12_A_configureMemory(ADC12_A_BASE, &param);
135
136     /* TTC/EPS regulator current sensor */
137     param.memoryBufferControlIndex          = ADC12_A_MEMORY_7;
138     param.inputSourceSelect                 = ADC12_A_INPUT_A6;
139     param.positiveRefVoltageSourceSelect    = ADC12_A_VREFPOS_EXT;
140     param.negativeRefVoltageSourceSelect    = ADC12_A_VREFNEG_AVSS;
141     param.endOfSequence                     = ADC12_A_NOTENDOFSEQUENCE;
142     ADC12_A_configureMemory(ADC12_A_BASE, &param);
143
144     /* Main bus voltage sensor */
145     param.memoryBufferControlIndex          = ADC12_A_MEMORY_10;
146     param.inputSourceSelect                 = ADC12_A_INPUT_A7;
147     param.positiveRefVoltageSourceSelect    = ADC12_A_VREFPOS_EXT;
148     param.negativeRefVoltageSourceSelect    = ADC12_A_VREFNEG_AVSS;
149     param.endOfSequence                     = ADC12_A_NOTENDOFSEQUENCE;
150     ADC12_A_configureMemory(ADC12_A_BASE, &param);
151
152     /* Solar Panel -Y/+X voltage sensor */
153     param.memoryBufferControlIndex          = ADC12_A_MEMORY_12;
154     param.inputSourceSelect                 = ADC12_A_INPUT_A12;
155     param.positiveRefVoltageSourceSelect    = ADC12_A_VREFPOS_EXT;
156     param.negativeRefVoltageSourceSelect    = ADC12_A_VREFNEG_AVSS;
157     param.endOfSequence                     = ADC12_A_NOTENDOFSEQUENCE;
158     ADC12_A_configureMemory(ADC12_A_BASE, &param);
159
160     /* Solar Panel -X/+Z voltage sensor */
161     param.memoryBufferControlIndex          = ADC12_A_MEMORY_13;
162     param.inputSourceSelect                 = ADC12_A_INPUT_A13;
163     param.positiveRefVoltageSourceSelect    = ADC12_A_VREFPOS_EXT;
164     param.negativeRefVoltageSourceSelect    = ADC12_A_VREFNEG_AVSS;
165     param.endOfSequence                     = ADC12_A_NOTENDOFSEQUENCE;
166     ADC12_A_configureMemory(ADC12_A_BASE, &param);
167
168     /* Solar Panel -Z/+Y voltage sensor */
169     param.memoryBufferControlIndex          = ADC12_A_MEMORY_14;
170     param.inputSourceSelect                 = ADC12_A_INPUT_A14;
171     param.positiveRefVoltageSourceSelect    = ADC12_A_VREFPOS_EXT;
172     param.negativeRefVoltageSourceSelect    = ADC12_A_VREFNEG_AVSS;
173     param.endOfSequence                     = ADC12_A_NOTENDOFSEQUENCE;
174     ADC12_A_configureMemory(ADC12_A_BASE, &param);
175
176     /* Solar Panel total voltage sensor */
177     param.memoryBufferControlIndex          = ADC12_A_MEMORY_15;
178     param.inputSourceSelect                 = ADC12_A_INPUT_A15;

```

```

179     param.positiveRefVoltageSourceSelect    = ADC12_A_VREFPOS_EXT;
180     param.negativeRefVoltageSourceSelect    = ADC12_A_VREFNEG_AVSS;
181     param.endOfSequence                     = ADC12_A_NOTENDOFSEQUENCE;
182     ADC12_A_configureMemory(ADC12_A_BASE, &param);
183
184     ADC12_A_clearInterrupt(ADC12_A_BASE, ADC12_A_IFG0 | ADC12_A_IFG1 |
185     ADC12_A_IFG2 | ADC12_A_IFG3 | ADC12_A_IFG4 | ADC12_A_IFG5 |
186     ADC12_A_IFG6 | ADC12_A_IFG7 | ADC12_A_IFG10 | ADC12_A_IFG12 |
187     ADC12_A_IFG13 | ADC12_A_IFG14 | ADC12_A_IFG15);
188
189     uint8_t i = 0;
190     for(i=0; i<ADC_TIMEOUT_MS; i++)
191     {
192         if (REF_ACTIVE != Ref_isRefGenBusy(REF_BASE))
193         {
194             break;
195         }
196
197         adc_delay_ms(1);
198     }
199
200     Ref_setReferenceVoltage(REF_BASE, REF_VREF1_5V);
201
202     Ref_enableReferenceVoltage(REF_BASE);
203
204     Ref_enableTempSensor(REF_BASE);
205
206     adc_delay_ms(10);
207
208     /* Temperature sensor calibration data */
209     TLV_getInfo(TLV_TAG_ADCCAL, 0, &adc_cal_bytes, (uint16_t **)&
210     adc_cal_data);
211
212     adc_mref = ((float)(adc_cal_data->adc_ref15_85_temp - adc_cal_data->
213     adc_ref15_30_temp)) / (85 - 30);
214     adc_nref = adc_cal_data->adc_ref15_85_temp - adc_mref * 85;
215
216     /* Mutex initialization */
217     adc_mutex_create();
218
219     adc_is_ready = true;
220
221     return 0;
222 }
223
224 int adc_read(adc_port_t port, uint16_t *val)
225 {

```

```
221     uint8_t i = 0;
222     for(i=0; i<ADC_TIMEOUT_MS; i++)
223     {
224         if (!ADC12_A_isBusy(ADC12_A_BASE))
225         {
226             break;
227         }
228
229         adc_delay_ms(1);
230     }
231
232     *val = 0xffff;
233     // Symbol 'UINT16_MAX' could not be resolved - Semantic Error
234     // *val = UINT16_MAX;
235
236     if (i == ADC_TIMEOUT_MS)
237     {
238         return -1; /* Timeout reached */
239     }
240
241     switch(port)
242     {
243         case ADC_PORT_0:
244             ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_0,
ADC12_A_SINGLECHANNEL);
245
246             for(i=0; i<ADC_TIMEOUT_MS; i++)
247             {
248                 if (ADC12_A_getInterruptStatus(ADC12_A_BASE,
ADC12_A_IFG0))
249                 {
250                     break;
251                 }
252
253                 adc_delay_ms(1);
254             }
255
256             if (i == ADC_TIMEOUT_MS)
257             {
258                 return -1; /* Timeout reached */
259             }
260
261             *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_0);
262
263             break;
264         case ADC_PORT_1:
```

```
265         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_1 ,
266         ADC12_A_SINGLECHANNEL);
267
268         for(i=0; i<ADC_TIMEOUT_MS; i++)
269         {
270             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
271             ADC12_A_IFG1))
272             {
273                 break;
274             }
275
276             adc_delay_ms(1);
277
278             if (i == ADC_TIMEOUT_MS)
279             {
280                 return -1; /* Timeout reached */
281             }
282
283             *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_1);
284
285             break;
286         case ADC_PORT_2:
287             ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_2 ,
288             ADC12_A_SINGLECHANNEL);
289
290             for(i=0; i<ADC_TIMEOUT_MS; i++)
291             {
292                 if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
293             ADC12_A_IFG2))
294             {
295                 break;
296             }
297
298             adc_delay_ms(1);
299
300             if (i == ADC_TIMEOUT_MS)
301             {
302                 return -1; /* Timeout reached */
303             }
304
305             *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_2);
306
307             break;
308         case ADC_PORT_3:
```

```
307         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_3 ,
ADC12_A_SINGLECHANNEL);
308
309         for(i=0; i<ADC_TIMEOUT_MS; i++)
310         {
311             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG3))
312             {
313                 break;
314             }
315
316             adc_delay_ms(1);
317         }
318
319         if (i == ADC_TIMEOUT_MS)
320         {
321             return -1; /* Timeout reached */
322         }
323
324         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_3);
325
326         break;
327     case ADC_PORT_4:
328         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_4 ,
ADC12_A_SINGLECHANNEL);
329
330         for(i=0; i<ADC_TIMEOUT_MS; i++)
331         {
332             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG4))
333             {
334                 break;
335             }
336
337             adc_delay_ms(1);
338         }
339
340         if (i == ADC_TIMEOUT_MS)
341         {
342             return -1; /* Timeout reached */
343         }
344
345         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_4);
346
347         break;
348     case ADC_PORT_5:
```

```
349         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_5 ,
ADC12_A_SINGLECHANNEL);
350
351         for(i=0; i<ADC_TIMEOUT_MS; i++)
352         {
353             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG5))
354             {
355                 break;
356             }
357
358             adc_delay_ms(1);
359         }
360
361         if (i == ADC_TIMEOUT_MS)
362         {
363             return -1; /* Timeout reached */
364         }
365
366         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_5);
367
368         break;
369     case ADC_PORT_6:
370         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_6 ,
ADC12_A_SINGLECHANNEL);
371
372         for(i=0; i<ADC_TIMEOUT_MS; i++)
373         {
374             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG6))
375             {
376                 break;
377             }
378
379             adc_delay_ms(1);
380         }
381
382         if (i == ADC_TIMEOUT_MS)
383         {
384             return -1; /* Timeout reached */
385         }
386
387         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_6);
388
389         break;
390     case ADC_PORT_7:
```

```
391         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_7 ,
ADC12_A_SINGLECHANNEL);
392
393         for(i=0; i<ADC_TIMEOUT_MS; i++)
394         {
395             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG7))
396             {
397                 break;
398             }
399
400             adc_delay_ms(1);
401         }
402
403         if (i == ADC_TIMEOUT_MS)
404         {
405             return -1; /* Timeout reached */
406         }
407
408         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_7);
409
410         break;
411     case ADC_PORT_8:
412         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_8 ,
ADC12_A_SINGLECHANNEL);
413
414         for(i=0; i<ADC_TIMEOUT_MS; i++)
415         {
416             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG8))
417             {
418                 break;
419             }
420
421             adc_delay_ms(1);
422         }
423
424         if (i == ADC_TIMEOUT_MS)
425         {
426             return -1; /* Timeout reached */
427         }
428
429         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_8);
430
431         break;
432     case ADC_PORT_9:
```



```
433         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_9 ,
ADC12_A_SINGLECHANNEL);
434
435         for(i=0; i<ADC_TIMEOUT_MS; i++)
436         {
437             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG9))
438             {
439                 break;
440             }
441
442             adc_delay_ms(1);
443         }
444
445         if (i == ADC_TIMEOUT_MS)
446         {
447             return -1; /* Timeout reached */
448         }
449
450         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_9);
451
452         break;
453     case ADC_PORT_10:
454         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_10 ,
ADC12_A_SINGLECHANNEL);
455
456         for(i=0; i<ADC_TIMEOUT_MS; i++)
457         {
458             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG10))
459             {
460                 break;
461             }
462
463             adc_delay_ms(1);
464         }
465
466         if (i == ADC_TIMEOUT_MS)
467         {
468             return -1; /* Timeout reached */
469         }
470
471         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_10);
472
473         break;
474     case ADC_PORT_11:
```

```
475         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_11 ,
ADC12_A_SINGLECHANNEL);
476
477         for(i=0; i<ADC_TIMEOUT_MS; i++)
478         {
479             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG11))
480             {
481                 break;
482             }
483
484             adc_delay_ms(1);
485         }
486
487         if (i == ADC_TIMEOUT_MS)
488         {
489             return -1; /* Timeout reached */
490         }
491
492         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_11);
493
494         break;
495     case ADC_PORT_12:
496         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_12 ,
ADC12_A_SINGLECHANNEL);
497
498         for(i=0; i<ADC_TIMEOUT_MS; i++)
499         {
500             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG12))
501             {
502                 break;
503             }
504
505             adc_delay_ms(1);
506         }
507
508         if (i == ADC_TIMEOUT_MS)
509         {
510             return -1; /* Timeout reached */
511         }
512
513         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_12);
514
515         break;
516     case ADC_PORT_13:
```

```
517         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_13 ,
ADC12_A_SINGLECHANNEL);
518
519         for(i=0; i<ADC_TIMEOUT_MS; i++)
520         {
521             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG13))
522             {
523                 break;
524             }
525
526             adc_delay_ms(1);
527         }
528
529         if (i == ADC_TIMEOUT_MS)
530         {
531             return -1; /* Timeout reached */
532         }
533
534         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_13);
535
536         break;
537     case ADC_PORT_14:
538         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_14 ,
ADC12_A_SINGLECHANNEL);
539
540         for(i=0; i<ADC_TIMEOUT_MS; i++)
541         {
542             if (ADC12_A_getInterruptStatus(ADC12_A_BASE ,
ADC12_A_IFG14))
543             {
544                 break;
545             }
546
547             adc_delay_ms(1);
548         }
549
550         if (i == ADC_TIMEOUT_MS)
551         {
552             return -1; /* Timeout reached */
553         }
554
555         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_14);
556
557         break;
558     case ADC_PORT_15:
```

```

559         ADC12_A_startConversion(ADC12_A_BASE, ADC12_A_MEMORY_15,
ADC12_A_SINGLECHANNEL);
560
561         for(i=0; i<ADC_TIMEOUT_MS; i++)
562         {
563             if (ADC12_A_getInterruptStatus(ADC12_A_BASE,
ADC12_A_IFG15))
564             {
565                 break;
566             }
567
568             adc_delay_ms(1);
569         }
570
571         if (i == ADC_TIMEOUT_MS)
572         {
573             return -1; /* Timeout reached */
574         }
575
576         *val = ADC12_A_getResults(ADC12_A_BASE, ADC12_A_MEMORY_15);
577
578         break;
579     default:
580         #if CONFIG_DRIVERS_DEBUG_ENABLED == 1
581             sys_log_print_event_from_module(SYS_LOG_ERROR,
ADC_MODULE_NAME, "Error reading the ADC port ");
582             sys_log_print_uint(port);
583             sys_log_print_msg("! Invalid port!");
584             sys_log_new_line();
585         #endif /* CONFIG_DRIVERS_DEBUG_ENABLED */
586         return -1;
587     }
588
589     ADC12_A_clearInterrupt(ADC12_A_BASE, ADC12_A_IFG0 | ADC12_A_IFG1 |
ADC12_A_IFG2 | ADC12_A_IFG3 | ADC12_A_IFG4 | ADC12_A_IFG5 |
ADC12_A_IFG6 | ADC12_A_IFG7 | ADC12_A_IFG10 | ADC12_A_IFG12 |
ADC12_A_IFG13 | ADC12_A_IFG14 | ADC12_A_IFG15);
590
591     return 0;
592 }
593
594 float adc_temp_get_mref(void)
595 {
596     return adc_mref;
597 }
598
599 float adc_temp_get_nref(void)

```

```
600 {  
601     return adc_nref;  
602 }  
603  
604 /** \} End of adc group */
```