UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO DE CIÊNCIAS, TECNOLOGIAS E SAÚDE

Alexandre Seemund Nicolás

AVALIAÇÃO DA CONFIABILIDADE DO SISTEMA OPERACIONAL FREERTOS EM NANOSSATÉLITES

Araranguá

2018

Alexandre Seemund Nicolás

AVALIAÇÃO DA CONFIABILIDADE DO SISTEMA OPERACIONAL FREERTOS EM NANOSSATÉLITES

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos necessários para obtenção do grau de bacharel em Engenharia de Computação.
Orientador: Prof. Dr. Anderson Luiz Fernandes Perez

Araranguá

2018

Alexandre Seemund Nicolás

AVALIAÇÃO DA CONFIABILIDADE DO SISTEMA OPERACIONAL FREERTOS EM NANOSSATÉLITES

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Engenharia de Computação", e aprovado em sua forma final pela Universidade Federal de Santa Catarina.

Araranguá, 05 de julho 2018.

Prof. Drª Eliane Pozzebon
Coordenadora

Banca Examinadora:

Prof. Dr. Anderson Luiz Fernandes Perez
Orientador

Prof. Dr. Marcelo Daniel Berejuck

Prof. Dr. Tiago Oliveira Weber

À minha mãe Marlene e meu irmão Michel.

AGRADECIMENTOS

Agradeço primeiramente a toda minha família por todo o carinho e apoio oferecido durante toda minha vida. À Universidade Federal de Santa Catarina pela oportunidade de poder estudar num lugar tão prestigiado e agradável, a todos os seus funcionários que me prestaram sempre um serviço de excelência. Ao Professor e orientador Anderson Luiz Fernandes Perez, pelo conhecimento compartilhado, pelas orientações, e principalmente por toda a sua atenção e paciência oferecida em todos esses anos. À todos os meus amigos que conheci na UFSC e que me ajudaram de alguma forma nesse período de formação acadêmica. A minha mãe Marlene Rosimar Seemund e meu irmão Michel Andreas Seemund Nicolás, por todo amor e afeto que me deram, e por sempre me apoiarem em meus sonhos e objetivos.

Saber muito não lhe torna inteligente. A inteligência se traduz na forma que você recolhe, julga, maneja e, sobretudo, onde e como aplica esta informação

Carl Sagan

RESUMO

Nanossatélites são pequenos corpos celestes artificiais que orbitam o planeta Terra em baixas altitudes. Por terem tamanho reduzido os nanossatélites carregam poucos instrumentos e são geralmente utilizados em experimentos científicos ou de monitoramento de fenômenos físico-químicos. Os nanossatélites estão se tornando populares em virtude do baixo custo de desenvolvimento comparado ao custo de desenvolvimento de um satélite. Um nanossatélite é composto por diversos componentes eletroeletrônicos e, por receber radiação cósmica, estes estão sujeitos a falhas. Por ser dotado de vários sensores e atuadores, a unidade CDH (do inglês, Command and Data Handling system) de um nanossatélite geralmente está equipada com um programa responsável pelo seu controle. Este é responsável por todo o gerenciamento das funções desempenhada no nanossatélite, desde a simples função de leitura de sensores e acionamento de atuadores, até funções mais críticas como as que ajustam rotas e provém comunicação. Este trabalho visa desenvolver um ambiente de testes para a validação do sistema operacional FreeRTOS em nanossatélites, com enfoque na unidade CDH.

Palavras-chave: nanossatélites, cubesat, tolerância a falhas, soft error, sistemas operacionais embarcados, unidade de comando e manipulação de dados.

ABSTRACT

Nanosatellites are small artificial celestial bodies that orbit the planet Earth at low altitudes. Because they are small in size, nanosatellites carry few instruments and are generally used in scientific experiments or in monitoring physicochemical phenomena. Nanosatellites are becoming popular because of the low cost of development compared to the cost of developing a satellite. A nanosatellite is composed of several electrical and electronic components and, for receive cosmic radiation, they are subject to failure. Because it is equipped with multiple sensors and actuators, the unit CDH (Command and Data Handling) of a nanosatellite is usually equipped with a program responsible for its control. This is responsible for all the management of the functions performed at the nanosatellite, from simple sensor reading and actuator actuation to more critical functions such as those that adjust routes and provide communication. This work aims to develop a testing environment for the validation of the FreeRTOS operating system in nanosatellite, focusing on the CDH unit. .

Keywords: nanosatellite, cubesat, fault tolerance, soft error, embedded operating system, command and data handling unit

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

Desde o sucesso do primeiro satélite enviado ao espaço pelo homem, o Sputnik I, a exploração espacial nunca mais parou. A tendência sempre foi criar satélites cada vez mais eficientes e com menor custo, consequentemente tornando estes menores e mais leves. Essa expansão teve um considerável crescimento nos últimos anos devido principalmente ao interesse em usar pequenos satélites em diferentes tipos de missões na órbita terrestre (NASA REPORT, 2015), em especial por um crescente mercado voltado à Cubesats, um padrão de satélite desenvolvido em 1999, que busca basicamente a redução de custo e de tempo de projeto, aumentando a frequência de lançamento (LEE, 2017).

Entretanto o ambiente espacial apresenta uma série de fenômenos que podem afetar o funcionamento dos dispositivos, principalmente sistemas computacionais constituídos por circuitos integrados COTS (*Commercial Off The Shelf*). Fenômenos como fluxos de partículas solares ou fluxos de radiação galáctica geram partículas ionizadas que, quando colidem com o dispositivo, produzem diferentes tipos de falhas. Não só sistemas espaciais estão optando por dispositivos de baixo custo oferecido pelo mercado, mas sistemas no geral críticos que precisam de segurança (especialmente em novas áreas do mercado, como automotivo, biomédico ou telecomunicação), e tem seu orçamento reduzido (NICOLAIDIS, 2010).

Para evitar os efeitos indesejados da radiação nos dispositivos, geralmente são utilizados dois tipos de abordagem, muitas vezes de forma conjunta: a prevenção e a tolerância a falhas. A prevenção de falhas por radiação é um método de construção e teste de componentes eletrônicos com o objetivo de torná-los resistentes aos danos e mau funcionamento causados pela radiação, assim a prevenção diminui a chances de ocorrer uma falha. A abordagem de tolerância ou robustez a radiação busca mitigar os efeitos da ocorrência de SEU (*Single Event Upset*) no sistema por meio de técnicas implementadas via hardware e/ou software, que faça com que o sistema tolere as falhas de componentes assim mantendo sua funcionalidade (MACHADO, 2014).

1.1 OBJETIVO GERAL

Avaliar a confiabilidade do sistema operacional FreeRTOS aplicado a uma unidade CDH de um nanossatélite.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos são:

a. Levantar o estado da arte a respeito de nanossatélites e cubesats;

b. Levantar o estado da arte sobre o ambiente espacial e o comportamento de dispositivos eletrônicos nele inseridos;

c. Fazer um estudo a respeito de sistemas operacionais embarcados;

d. Estudar a arquitetura e a API (*Application Programming Interface*) de programação do sistema operacional FreeRTOS;

e. Avaliar o FreeRTOS em um ambiente que simule as condições e requisitos de um nanossatélite real;

f. Analisar e descrever os resultados obtidos em (e).

1.3 MOTIVAÇÃO E JUSTIFICATIVA

A exploração espacial sempre foi muito custosa e demanda um tempo de projeto extenso, devido a sua complexidade. Esse cenário está mudando, e a busca por opções mais baratas e de fácil desenvolvimento vem sendo um dos principais requisitos no desenvolvimento espacial, principalmente com o surgimento do padrão de projetos para Cubesats. Cubesats são pequenos satélites com tamanho e massa definidos, tem como principal objetivo diminuir o custo e o tempo de desenvolvimento de nanossatélites, consequentemente aumentando sua frequência de lançamento e acessibilidade ao espaço. O que era antes feito apenas por agências espaciais especializadas, agora é realizado por estudantes universitários e empresas voltadas para o comércio em geral.

A evolução das tecnologias utilizadas em satélites ajudou também no processo de miniaturização de seus componentes, deixando eles mais leves e com menor consumo energético. Fabricantes da área aviônica estão produzindo sistemas cada vez mais integrados. O que era antes desenvolvido apenas por demanda, agora é produzido comercialmente e de fácil acesso.

Um sistema que promova uma grau de confiabilidade elevado é de grande importância no meio espacial, e se desenvolvido a partir do zero, pode causar atrasos de projetos e ser custoso. Cubesats geralmente usam sistemas operacionais embarcados em sua arquitetura, eles

promovem um gerenciamento das tarefas, contendo prazos e prioridades a serem cupridas.

O FreeRTOS é um sistema operacional de tempo real, no qual foi desenvolvido para atender requisitos de tempo e de gerenciamento de tarefas. Sua utilização no mercado vem crescendo, principalmente em sistemas embarcados, pela sua portabilidade e especialmente por ser confiável. É de código aberto e de fácil desenvolvimento.

1.4 METODOLOGIA

O sistema será desenvolvido na linguagem de programação C, com o compilador XC16 para a família de microcontroladores PIC24F. O programa usado para a avaliação será desenvolvido na IDE (Ambiente Integrado de Desenvolvimento, do Inglês, *Integrated Development Environment*) MPLAB da Microchip. Serão realizados os experimentos no *debugger* do MPLAB.

1.5 ORGANIZAÇÃO DO TRABALHO

Este trabalho está dividido em 6 capítulos, contando com a introdução.

O **Capítulo 2** aborda alguns conceitos fundamentais para entender o que é um satélite, como funciona e como este se comporta no espaço. Apresenta algumas características de pequenos satélites, explicando sobre o conceito de CubeSats e nanossatélites em geral. Menciona alguns fenômenos que ocorrem em dispositivos eletrônicos quando estes estão no espaço, apresentando quais componentes são mais afetados. Ao final do capítulo são apresentados os principais métodos para prevenir e reverter os possíveis erros causados por fenômenos do ambiente espacial.

O **Capítulo 3** apresenta alguns conceitos básicos de sistemas operacionais, características, tipos e aplicações. O capítulo também discorre sobre os sistemas operacionais embarcados (*embedded system*) e os sistemas operacionais de tempo real (*real time system*).

O **Capítulo 4** apresentada a arquitetura e as principais características do sistema de comando e tratamento de dados desenvolvido para avaliar o FreeRTOS.

O **Capítulo 5** apresenta os resultados obtidos dos testes gerados para garantir a confiabilidade do FreeRTOS aplicado em unidades

CDH em nanossatélites. Foram realizados 5 experimentos neste sistema, variando a quantidade de carga e a quantidade de erros gerados, nas diferentes partes da memória, formando assim um conjunto de informações que são utilizados na análise de confiabilidade do FreeRTOS para nanossatélites.

O **Capítulo 6** apresenta as considerações finais e algumas propostas para trabalhos futuros.

2 FUNDAMENTOS DE NANOSSATÉLITES

Este capítulo aborda alguns conceitos fundamentais para entender o que é um satélite, como funciona e como este se comporta no espaço. Serão apresentadas algumas características de pequenos satélites, explicando sobre o conceito de CubeSats e nanossatélites em geral. Serão mencionados alguns fenômenos que ocorrem em dispositivos eletrônicos quando estes estão no espaço, apresentando quais componentes são mais afetados. E no final deste capítulo será mostrado também os principais métodos para prevenir e reverter os possíveis erros causados por esses fenômenos do ambiente espacial.

2.1 SATÉLITES

A primeira pessoa a empregar a palavra latina *satelles* foi Galileu, para descrever as luas de Júpiter. *Satelles* significa servo, guarda ou atendente de um mestre poderoso ou senhor na Roma antiga, um *satelles* circulava a cidade para atender as ordens de seu mestre e para oferecer proteção à casa que servia. Hoje, o termo satélite é empregado a qualquer objeto que tem sua órbita em outro maior e de massa superior, a lua é um satélite que orbita a terra, e a terra é um satélite que orbita o sol. Terra e lua neste caso são satélites naturais. Qualquer artefato espacial feito pelo homem que possui sua órbita ao redor de algum corpo celeste é chamado de satélite artificial.

As forças que mantém os satélites em torno da terra são as mesma que atuam em todos os corpos celestes. O primeiro a estudar o movimento dos corpos celestes, foi Johannes Kepler baseando seus estudos nos dados observados pelo o astrônomo Tycho Brahe. Kepler escreveu três leis do movimento planetário, datadas entre 1609 à 1619. Essas leis são descritas da seguinte forma:

- **Primeira lei:** a órbita de cada planeta é uma elipse tendo o Sol por um dos focos.
- **Segunda lei:** a linha que une o planeta ao Sol varre áreas iguais em tempos iguais.
- **Terceira lei:** o quadrado do período orbital de um planeta é proporcional ao cubo de sua distância média ao Sol.

Isaac Newton enunciou a lei da gravitação universal em 1687,

cujo resultado é fundamental para entender o movimento orbital entre os planetas e objetos que o rodeiam. Newton afirma que a força de atração é inversamente proporcional ao quadrado da distância entre os objetos e diretamente proporcional ao produto de suas massas. A força gravitacional F(g) é dada pela seguinte equação:

$$Fg = \frac{GMm}{R^2} \tag{2.1}$$

onde, $G = 6,673.10^{-20} km^3/kg.s^2$ é a constante universal de gravitação, M é a massa da Terra (ou de um corpo celeste, dependendo do caso), m a massa do satélite e r é a distância do centro das massas M e m. E para se determinar intensidade da força centrípeta (Fc) atuante nos satélites tem-se a seguinte fórmula:

$$Fc = \frac{mv^2}{R} \tag{2.2}$$

Além dessas equações e leis mencionadas por Kepler e Newton, existem uma série de outras essenciais para poder enviar e direcionar um satélite com sucesso para o espaço. Geralmente os satélites são levados por foguetes, sendo liberados ao chegar em seu destino, então eles são ejetados para sua órbita final. Às vezes o motor do foguete é utilizado para deslocar o satélite para sua órbita final. Tudo isso é monitorado em centros de controle, que recebem dados constantes da posição e trajetória do satélite a cada instante. As operações espaciais são caras, complexas e demandam monitoramento severo, por isso foi desenvolvido o padrão de CubeSat para pequenos satélites, diminuindo custo e aumentando sua frequência de lançamentos.

Um marco importante na história espacial e principalmente para a humanidade, foi o sucesso do lançamento ao espaço do primeiro satélite artificial do mundo, o Sputnik I (Figura 1). Desenvolvido pela União Soviética, tinha o tamanho de uma bola de praia com 58 cm de diâmetro, pesava apenas 83,6 kg e levou cerca de 98 minutos para orbitar a Terra . Um famoso evento que marcou o início da corrida espacial entre Estados Unidos e União Soviética (GARBER, 2017), e consequentemente a era espacial. Desde então a evolução científica e tecnológica não parou mais, trazendo consigo grandes mudanças no cenário mundial e espacial.

Figura 1 – Foto do Sputnik I

Extraído de: *www.newworldencyclopedia.org/entry/Spaceexploration*

2.2 NANOSSATÉLITES

Desde o sucesso do primeiro satélite enviado ao espaço pelo homem, a exploração e desenvolvimento espacial nunca mais parou, trazendo consigo desenvolvimento militar, científico e tecnológico. A tendência sempre foi criar satélites cada vez mais eficientes e com menor custo, consequentemente tornando eles menores e mais leves. Essa tendência nos levou ao desenvolvimento de mini, micro e até mesmo nanossatélites, onde seus pesos e tamanhos variam de toneladas e metros à gramas e centímetros. O termo "nanossatélite" está sendo utilizado para classificar uma categoria de pequenos satélites, essa nomenclatura designa a faixa de massa que determinado satélite possui. A Tabela 1 relaciona nomenclatura com a massa do satélite (MARINAN, 2013).

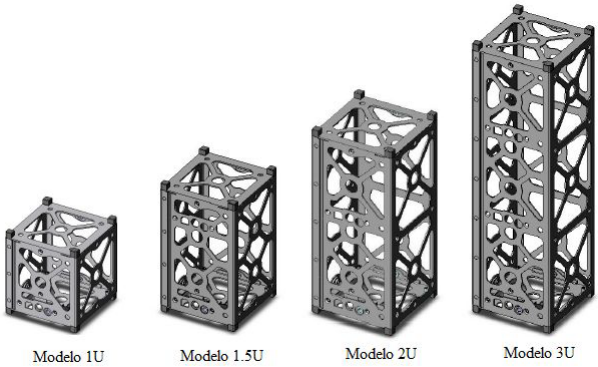
Tabela 1 – Nomenclatura para Satélites

Nomenclatura	Peso(kg)
Minissatélite	+100
Microssatélite	10 - 100
Nanossatélite	1 - 10
Picossatélite	0.01 - 1
Femtossatélite	0.001 - 0.01

2.2.1 CubeSats

Em 1999 os professores Jordi Puig-Suari da Universidade Politécnica Estadual da Califórnia (Cal Poly), e Bob Twiggs do Laboratório de Desenvolvimento de Sistemas Espaciais (SSDL) da Universidade de Stanford, criaram um padrão de projeto para nanosatélites, cujo nome seria especificado como CubeSat. Esse padrão tinha como principal objetivo diminuir custo e tempo de desenvolvimento de nanossatélites, consequentemente aumentaria sua frequências de lançamentos e acessibilidade ao espaço (LEE, 2017). Segundo as especificações, os Cubesat teriam uma unidade de tamanho padrão (U), que seria um cubo de 10 x 10 x 10 cm de lado. E seus tamanho podem variar de 1U, 1.5U, 2U e 3U, como ilustra a Figura 2.

Figura 2 – Tamanho por Unidade (U)



Extraído de: Lee (2017)

2.3 COMPONENTES DE UM SATÉLITE

Esta seção apresenta um resumo das partes que estão mais presentes em pequenas espaçonaves e CubeSats, evidenciando o funcionamento e as tecnologias mais utilizadas em sua estrutura.

2.3.1 Sistema Elétrico de Potência

A estrutura que garante o armazenamento, produção e gerenciamento de toda a energia de um nanossatélite é o EPS, do inglês *Electrical Power System*, que compõe aproximadamente um terço da massa total da espaçonave. As tecnologias de geração de energia incluem células fotovoltaicas, painéis solares, geradores termoelétrico de radioisótopos e geradores de energia termonuclear.

Cada EPS contém o sistema de gerenciamento e distribuição de energia - *Power Management And Distribution* (PMAD) - este permite que os operadores controlem o fluxo de energia para os instrumentos e subsistemas da espaçonave. Os sistemas PMAD possuem várias formas e costumam ser personalizados para atender a requisitos específicos da missão. Os engenheiros geralmente se concentram nas tecnologias de geração e armazenamento de energia que possuem uma relação de alta potência fornecida para uma pequena massa, esse subsistema geralmente é dado em energia específica (Wh/kg).

A tecnologia mais utilizada em pequenos satélites para produção de energia são as células fotovoltaicas ou células solares. Essas são feitas de um material semicondutor capaz de converter a luz proveniente do sol (energia solar) diretamente em energia elétrica por intermédio do efeito fotovoltaico. A geração de energia proveniente por essas células cai com o quadrado da distância do Sol, e varia com o cosseno do ângulo entre o painel e o raio solar.

2.3.2 Propulsão

Atualmente existe uma ampla variedade de tecnologias para sistemas de propulsão, porém a miniaturização desses sistemas para pequenas espaçonaves e principalmente nanossatélites é um grande desafio. Vale ressaltar que deve-se sempre levar em consideração o propósito da missão, assim considerar tipos de manobras, precisão e velocidade.

É possivel classificar os propulsores em duas categorias, sistemas

de propulsão química e sistemas de propulsão elétrica. A Tabela 2 descreve as tecnologias mais utilizadas para propulsão espacial, mostrando as duas grandezas mais relevantes da mesma, o impulso específico e empuxo. O impulso específico é a quantidade de impulso que pode ser produzida usando uma unidade de combustível, esta é a característica mais importante do método de propulsão, pois determina a velocidade máxima que pode ser obtida. O empuxo é a quantidade média de impulso dado em segundos.

Tabela 2 – Tipos de propulsão

Sistema	Impulso Específico	Empuxo
Hidrazina	0.5 - 4N	150 - 250s
Gás Frio	10mN - 10N	65 - 70s
Propulsão Não Tóxica	0.1 - 27N	220 - 250s
Propulsor de Plasma	1-1300uN	500 - 3000s
Propulsores de Electrosprays	10-120uN	500 - 5000s
Propulsores de Efeito Hall	10-50mN	1000 - 2000s
Motores de íon	1-10nN	1000 - 3500s
Vela Solar	0.25-0.6mN	N/A

2.3.3 Orientação, Navegação e Controle

É o subsistema responsável pela determinação da posição e controle de altitude do nanossatélite. Para os satélites de baixa órbita, os receptores de GPS (*Global Positioning System*) são o principal método para a determinação de órbita, substituindo os métodos de rastreamento baseados no solo. Os receptores de GPS são considerados uma tecnologia madura, principalmente para naves espaciais pequenas. Para aplicações mais específicas existem unidades integradas que combinam vários componentes em uma única peça, contendo magnetômetro, magnetorque, rodas de reação e *star trackers*. Essas unidades geralmente incluem algoritmos de gerenciamento de *momentum* e determinação de atitude incorporados. Para determinar a posição no espaço profundo, é utilizado o *Deep Space Network* (DSN), junto com um rádio transponder.

2.3.4 Comando e Tratamento de Dados

Existe uma demanda de satélites do tipo Cubesats para cumprir missões mais complexas e com maiores requisitos operacionais, principalmente para missões que ultrapassem a LEO (*Low Earth Orbit*) e entram no espaço profundo. Esse aumento de requisitos influencia diretamente no subsistema de comando e tratamento de dados, do inglês *Command and Data Handling System* (CDH). O subsistema CDH atua como o "cérebro" do satélite e requer uma maior confiabilidade e melhor desempenho, principalmente para pequenos satélites, pelo uso de sistemas altamente integrados e pela necessidade de eficiência energética e de massa. Ainda assim são sistemas de fácil desenvolvimento e de baixo custo. Seu hardware inclui o computador de bordo e o software que controla as operações do satélite.

Atualmente vários Cubesats já foram lançados com sucesso no ambiente LEO, com missões de duração curta, normalmente de menos de um ano. Na maioria desses Cubesats foi utilizado sistemas com componentes COTS, onde existe uma crescente variedade de companhias e fornecedores comerciais, voltadas principalmente para o desenvolvimento de Cubesats e indústria espacial em geral. Esses fornecedores comerciais conseguem oferecer sistemas altamente integrados, contendo computador de bordo, memória, sistema elétrico de potência (EPS) e uma capacidade de suportar uma grande variedade de entradas e saídas (I/O) para a classe de Cubesat. Para missões de longa duração que ultrapassem a LEO, os vendedores estão incorporando design de projetos onde exista proteção ou tolerância contra radiação.

As FPGAs (*Field-Programmable Gate Array*) têm um legado de sucesso no espaço em relação a microcontroladores, e continuam oferecendo altos níveis de integração, fornecendo periféricos, memórias e melhor desempenho energético, fatores que influenciam na escolha do computador *on-board* atualmente. Muitos microcontroladores também tem se mostrado eficientes em termos energético e vem sendo usados em Cubesats, geralmente possuem processador ARM (*Advanced RISC Machine*) e uma variedade de periféricos *on-chip*, como USB (*Universal Serial Bus*), CAN (*Controller Area Network*)e interfaces como I2C (*Inter-Integrated Circuit*) e SPI (*Serial Peripheral Interface*). Houve também um aumento no número de microcontroladores que integram memória flash, por suas vantagens na programabilidade.

Existem vários hardwares de código aberto desenvolvido para Cubesats. A nave espacial ArduSat usou a plataforma Arduino para sua construção que usa microcontroladores da Atmel onde os desenvol-

vedores podem explorar seu ambiente de código aberto para escrever o software. O BeagleBone também surgiu como uma popular plataforma de código aberto, e contém um processador ARM que suporta o OpenCV. O BeagleSat é uma plataforma de cubesats de código aberto baseada na placa de desenvolvimento embarcado do BeagleBone, fornecendo um conjunto de ferramentas para projetar um cubesat a partir do zero. O Raspberry Pi é outro hardware de código aberto de alto desempenho, capaz de lidar com imagens e, potencialmente, com aplicativos de comunicação de alta velocidade. A Intel também entrou no mercado com seu sistema Edison, com o dual-core x86-64 SoC, a princípio foi direcionado para aplicações "*Internet of Things*", mas o Edison provou ser muito adequado para o desenvolvimento avançado de Cubesats. Existem muitos fabricantes que fornecem uma variedade de componentes eletrônicos que são considerados de alta confiabilidade no espaço. A Tabela 3 lista alguns desses fabricantes de sistemas *On-board* de computadores.

Tabela 3 – Exemplo de alguns Fabricantes de Sistemas *On-board*

ATMEL	Honeywell	STMicroelectronics
BAE Systems	Intel	Texas Instruments
Broadreach	Intersil	3D Plus
C-MAC Microtechnology	Maxwell Technologies	Xilinx
Cobham (Aeroflex)	Microsemi (Actel)	Arduino
Freescale	Space Micro, Inc.	BeagleBone

Existe uma grande variedades de memórias para naves espaciais pequenas, elas tipicamente começam com tamanho em torno de 32kB e aumentam de acordo com a tecnologia. Para funções do CDH, a memória requer alta confiabilidade. Encontra-se no mercado memórias com diferentes tipos de tecnologias desenvolvidas com características específicas, incluindo memória estática de acesso aleatório (SRAM), memória dinâmica de acesso aleatório (DRAM), memória Flash (tipo de memória programável de somente leitura apagável eletricamente), magnetoresistiva RAM (MRAM), ferro-elétrica RAM (FERAM), calcogeneto RAM (CRAM) e memória de mudança de fase (PCM). A SRAM é normalmente usada devido ao preço e disponibilidade.

Em missões no espaço profundo ou na LEO de longas durações, é preciso que os desenvolvedores incorporem a mitigação de radiação

em seus respectivos projetos. Cubesats tem tradicionalmente utilizado componentes COTS em seus projetos, o uso dessas peças permitiu um desenvolvimento de CDH de baixo custo. Muitos dos fornecedores de componentes e sistemas também fornecem dispositivos resistentes à radiação (*rad-hard*). Embora existam muitos componentes *rad-hard* comercialmente disponíveis, o uso desses componentes tem um impacto nos custos gerais do desenvolvimento de naves espaciais. Para manter os custos o mais razoável possível, os desenvolvedores da CDH precisam cancelar o uso dos componentes *rad-hard* e utilizar outras técnicas de mitigação de radiação, tornando o projeto tolerante à radiação.

Para aplicações espaciais, a radiação pode afetar os componentes eletrônicos de duas maneiras. A dose ionizante total (*Total Ionizing Dose* - TID) é a quantidade de radiação que o dispositivo recebe cumulativamente, é medida em krad e pode afetar o desempenho do transistor. O outro é chamado de efeito de evento único (*Single Event Effects* - SEE), é o distúrbio criado pelas partículas individuais que atingem a eletrônica. Os SEEs são classificados em outras subclasses de acordo com a forma que ele reage ao componente eletrônico, e será explicado com mais detalhes posteriormente. A Tabela 4 compara os vários tipos de memórias com suas características. Cabe resaltar que a escolha da memória está relacionada também a quantidade de radiação que é resistente, por isso valores de TID e SEU (*Single Event Upset*) *rate* devem ser levados em consideração.

Tabela 4 – Comparação de Tipos de Memórias

Tipo	SRAM	DRAM	Flash	MRAM	FERAM	CRAM/PCI
Não-Volátil	Não	Não	Sim	Sim	Sim	Sim
Faixa de Tensão	3.3-5V	3.3V	3.3-5V	3.3V	3.3V	3.3V
Retenção de Dados	N/A	N/A	10 anos	10 anos	10 anos	10 anos
Tempo de Acesso	10ns	25ns	50ns	300ns	300ns	100ns
Radiação (TID)	1Mrad	50krad	30krad	1Mrad	1Mrad	1Mrad
SEU *rate* (relativo)	Baixo-nulo	Alto	Nulo	Nulo	Nulo	-
Potência	500mW	300mW	30mW	900mW	270mW	-

2.4 AMBIENTE ESPACIAL

Junto com a exploração espacial, veio uma série de anomalia que ocorreram em seus dispositivos eletrônicos ocasionando falhas, supostamente proveniente do ambiente espacial. O primeiro registro de falhas,

evidenciando pelo efeito radioativo, foi relatado por May e Woods da Intel Corporation (NICOLAIDIS, 2010, p. v) relatando que partículas alpha induziram um "*soft errors*" na série 2107 de 16-KB DRAMS. Eles foram os primeiros a usar o termo "*soft errors*" para erros induzido por radiação, em meados dos anos 60. Existem outros registros que descrevem anomalias no sistema provocadas por radiação ionizada. Em 1975, Binder et al. relataram mau funcionamento do circuito, sendo provocado por uma "anomalia". Raios cósmicos de alta energia colidiam em regiões sensíveis do material semicondutor dos dispositivos eletrônicos do satélite, produzindo um acúmulo de cargas por trilhas de ionização, mudando o estado lógico dos flip-flops. Outro caso em 1979, Ziegler e Lanford presumiram que *soft errors* aumentariam à medida que a altitude aumentasse. De acordo com eles, raios cósmicos interagiam com o material do chip, degradando o núcleo de silício. Futuramente a IBM confirmou essa hipótese, reportando erros devidos a raios cósmicos em dados coletados, que estavam armazenados em um computador de sua empresa (TORRES, 2013).

Desde então, *soft erros* e seus efeitos em circuitos eletrônicos vem sendo estudado e está se tornando um dos problemas mais desafiadores encontrados em sistemas modernos atuais.

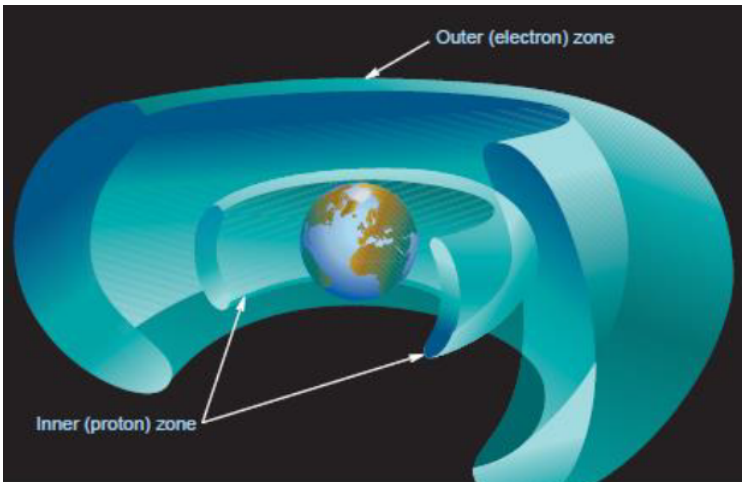
2.5 RADIAÇÃO ESPACIAL E SEUS EFEITOS

Existem alguns tipos de fontes de radiação e partículas que são mais nocivas aos componentes e circuitos eletrônicos. São duas as categorias para classificar a radiação, ionizante e não ionizante. Esses dois tipos fazem parte do ambiente espacial, as ionizantes podem emitir e propagar energia no espaço, podendo ser gerada por ondas ou partículas subatômicas, fazem parte dessa categoria os raios cósmicos, raios-X e radiação provenientes de matérias radioativas. A radiação não ionizante são incapazes de ionizar um material, sendo estas a luz ultravioleta, ondas de rádio e micro-ondas (TORRES, 2013, p. 9).

Chegando na atmosfera terrestre, os fluxos de partículas solares e fluxos de radiação galáctica, ao colidirem com núcleos de átomos da atmosfera terrestre, produzem chuveiros de partículas secundárias como nêutrons, píons, elétrons, fótons e prótons. Essas partículas, dependendo da sua energia ou massa, interagem de diversas formas com as moléculas da atmosfera terrestre, perdendo energia até serem finalmente freadas ou absorvidas na atmosfera (MACHADO, 2014). Os efeitos destas partículas está ligado diretamente a quão longe o componente

eletrônico está em relação a terra, isso porque a atmosfera terrestre age como um filtro natural que reduz a intensidade da radiação proveniente do espaço, e essa intensidade varia conforme a altitude em relação a terra. Isso é devido à interação do campo magnético da terra com ventos solares, formando o chamado de cinturões de Van Allen. A Figura 3 ilustra este fenômeno. A parte mais interna do cinturão está a uma distância de 100km a 10.000km da terra, e externa do cinturão situa-se entre 20.000km até aproximadamente 60.000km de altitude (BALEN, 2010).

Figura 3 – Cinturões de Van Allen

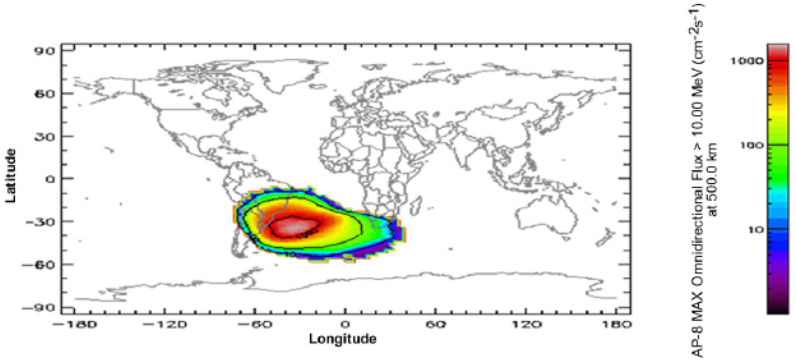


Fonte: Machado (2014).

Uma outra característica importante em relação a magnetosfera terrestre é a Anomalia Magnética do Atlântico Sul (AAS), onde existe uma deformação nas linhas de campo magnético sobre o sul do Brasil, formando uma espécie de depressão. Isso existe devido a uma diferença entre o centro do dipolo magnético e o centro geográfico da Terra, nessa região os cinturões de radiação alcançam menores altitudes, penetrando nas camadas mais baixas da atmosfera. A Figura 4 ilustra uma simulação realizada pela agência espacial européia SPENVIS (*Space Environment Information System*), utilizando o modelo AP-8 para o fluxo de prótons em uma altitude de 500 km na região do atlântico sul. A Figura 6 ilustra a influência do campo magnético terrestre na distribuição

de prótons no globo, é possível observar que o fluxo de prótons se concentra apenas na região do atlântico sul, sendo desprezível a presença deste no restante do planeta (TORRES, 2013).

Figura 4 – Anomalia do Atlântico Sul



Fonte: TORRES (2013).

É possível classificar em três tipos básicos de efeitos induzidos por radiação que afetam os circuitos eletrônicos.

2.5.1 Total Ionization Dose (TID)

Dose Ionizante Total - TID, é um efeito de carácter acumulativo que provoca a degradação do material do circuito, devido a exposição prolongada deste à radiação ionizante, pode ser medida em termos de J/kg ou rad. Esse acúmulo provoca diferentes alterações no dispositivo, como aumento no consumo de energia ou até mesmo perda do componente (TORRES, 2013).

2.5.2 Displacement Damage (DD)

Danos por deslocamento - DD, danos físicos na estrutura cristalina do material, geralmente silício, causando danos na estrutura cristalina de um material pela colisão, elástica ou não, de uma partícula de alta energia, alterando as características eletrônicas do componente. Também chamada de Dose Total Não Ionizante (*Total Non Ionization*

Dose), de forma a distinguir este fenômeno do TID (BALEN, 2010).

2.5.3 Single Event Effects (SEEs)

Efeitos de evento único - SEE, são um grupo de fenômenos com efeitos estocásticos que produzem falhas no funcionamento dos componentes eletrônicos, alterando um sinal ou danificando permanentemente o dispositivo. São causados pelo impacto de partículas ionizadas com o silício e com as demais impurezas presentes nos dispositivos, ionizando o material semicondutor e podendo provocar um pulso de corrente transiente por um intervalo de tempo indeterminado. Os SEEs podem ser classificados como:

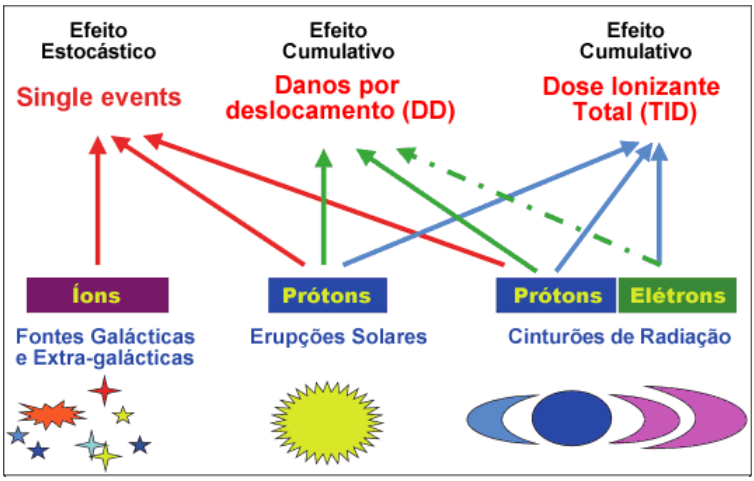
- *Single Event Upset (SEU):* modifica o estado lógico da memória em um bit armazenado. É causado quando uma partícula carregada colide diretamente com uma região sensível de uma célula de memória, conforme a quantidade de energia liberada pela partícula, o valor armazenado na memória pode ser invertido. Esse efeito também é chamado de *bit-flip*. Segundo Nicolaidis (2010) o termo SEU é utilizado, muitas vezes, como sendo o conjunto de SEEs não destrutivos.

- *Single Event Transient (SET):* efeito que gera um pulso transiente que pode ou não ser capturado por um elemento de memória (registradores), gerando um valor lógico errado. Ocorre tanto em dispositivos digitais, quanto em circuitos analógicos.

- *Hard Errors:* é uma categoria de eventos destrutivos, que danificam permanentemente o circuito. Estão nessa categoria os SEL (*Single Event Latchup*), SEB (*Single Event Burnout*) e SEGR (*Single Event Gate Rupture*).

Na Figura 5 é ilustrado a relação entre as diversas fontes de radiação existente no espaço e seus efeitos.

Figura 5 – Fontes de radiação espacial e seus efeitos



Retirada de: TORRES (2013).

2.6 EFEITO DA RADIAÇÃO EM DISPOSITIVOS ELETRÔNICOS

Existem alguns dispositivos considerados mais vulneráveis a efeitos da radiação, há poucos anos os elementos de circuitos considerados mais suscetível a *soft error* eram as DRAMs, e em segundo plano as SRAMs. As SRAMs eram mais robustas, tinham transistores de *pull-up* e *pull-down* que estabilizavam as cargas que representavam o estado da memória. Entretanto suas tecnologias de fabricação mudaram, DRAMs tornaram-se cada vez mais robusta a *soft error*, e as SRAMs mais vulneráveis (NICOLAIDIS, 2010, p. 10).

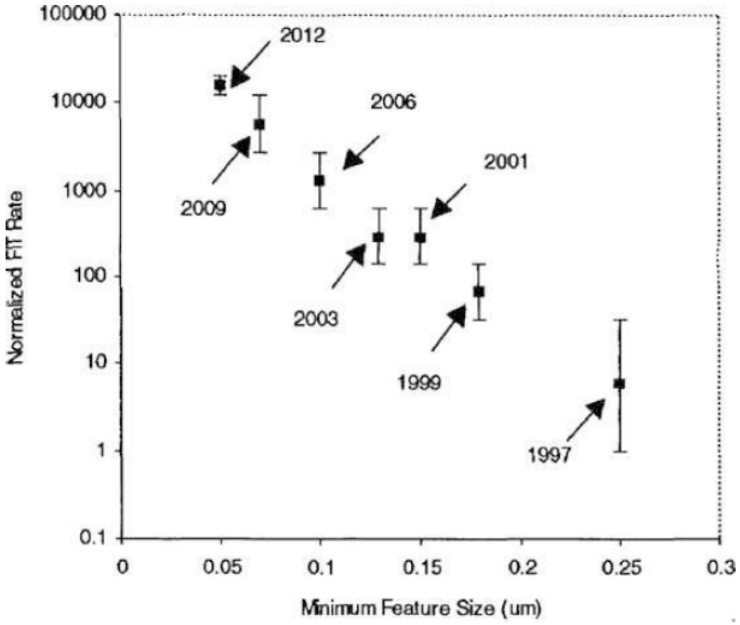
2.6.1 SRAMs

As SRAMs eram significativamente mais robustas contra *soft error* induzidos por radiação do que as DRAMs. Isso ocorreu porque, em uma célula de SRAM, os dados são armazenados em um loop de *feedback* de dois inversores de acoplamento cruzado. Este loop de *feedback* força a célula de bits a permanecer em seu estado programado.

No entanto, com o avanço da tecnologia, a tensão de alimentação e a capacitância diminuíram, o que resultou em uma menor carga crítica (Qcrit) em cada geração de SRAM.

Cohen et al. (1999), publicou um dos primeiros estudos sobre as tendências de *soft error*. Ele usou uma taxa conhecida como SER (*Soft Error Rate*), essa taxa de falhas é relatada em FIT[1] ou FIT/Mbit. O estudo determinou experimentalmente que o SER aumenta exponencialmente com a diminuição da tensão de alimentação das memórias (COHEN T.S. SRIRAM; FLATLEY, 1999). Eles estimaram que o SER aumentaria quase 100x em uma década. A Figura 6 contém o gráfico com essa tendência de valores de SER para os próximos anos.

Figura 6 – Tendência para FIT/Mbits em SRAM



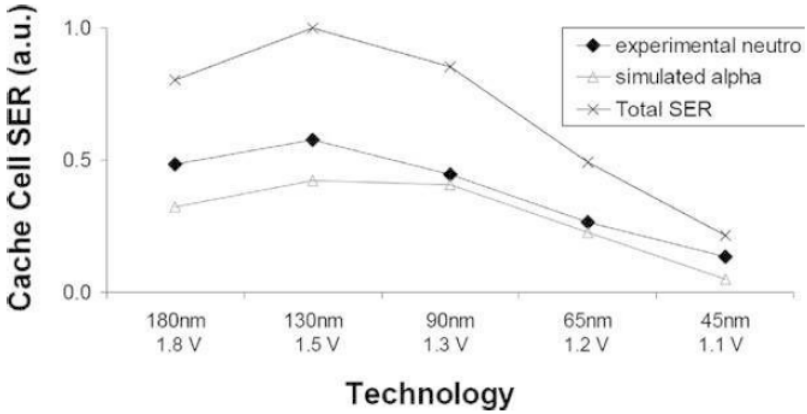
Retirada de: Cohen T.S. Sriram e Flatley (1999).

Seifert et al. (2006) da Intel, mostrou experimentalmente um

[1]FIT (*Failure In Time*) é uma unidade usada para mensurar quantidades de falhas em um determinado tempo, onde 1 FIT denota uma falha por bilhão de horas em um dispositivo (ou seja, uma falha em 114,077 anos).

resultado diferente de Cohen et al. (1999). O SER aumenta até uma determinada faixa de de tensão (1,5V) e satura conforme a tensão diminui, como ilustra a Figura 7. Isso ocorre por que a taxa de erro é proporcional também ao tamanho da memória, e não somente da tensão de alimentação. Com a redução da dimensão da memória, existe a diminuição da probabilidade de uma partícula acertar o bit da célula da memória. Para valores de tamanhos menores que 130nm, o SEU diminui (SEIFERT P. SLANKARD, 2006).

Figura 7 – Tendências para SER por bit em caches de SRAM



Retirada de: Seifert P. Slankard (2006)

Com a redução contínua das dimensões das memórias, ocorreu o aumento da probabilidade de uma única partícula causar MCU[2]. Com o avanço da tecnologia a porcentagem de eventos MCU está aumentando rapidamente e pode ser algumas dezenas de vezes quando o tamanho da célula de memória atinge valores entre 65 e 45 nm. São conhecidos dois mecanismos que podem produzir MCUs. No primeiro, as taxas induzidas por radiação podem ser compartilhadas por células de bits vizinhas. No segundo, a carga injetada pode desencadear um transistor bipolar parasita, o que resulta em um aumento na quantidade de carga coletada por essas células de bits vizinho. No caso de amplificação

[2]MCU (*Multiple-Cell Upset*) é quando SEU induz vários bits de um IC a falhar ao mesmo tempo.

bipolar, também conhecido como "efeito de bateria", a taxa de MCU aumenta com o aumento da tensão de alimentação, enquanto que no caso de compartilhamento de carga, a taxa de MCU diminui se a tensão for aumentada. Em geral, a taxa de MCU relativa é maior para nêutrons do que para partículas alfa porque no nêutron são geradas mais cargas em silício e também porque um nêutron atingindo um átomo em um dispositivo semicondutor muitas vezes o que produz múltiplas partículas ionizantes.

2.6.2 DRAMs

Nas memórias DRAM produzidas por tecnologias de processo CMOS, os nêutrons têm maior interferência quando comparado a partículas alfa, quando se fala em principal causador de *soft error* nas memórias dinâmicas de acesso aleatório.

À medida que a tecnologia se reduz a uma escala abaixo de 100 nm, as células de bits de memória estão se tornando mais resistentes a perturbações causadas por ambiente de radiação. Isso ocorre porque a capacitância das células de bits não está diminuindo com o avanço da tecnologia. Mesmo com a utilização de capacitores 3D com áreas de junção ainda menores que os 2D, estes contribuem para um SER ainda mais baixo, além de melhorar o desempenho das DRAMs. Como a capacitância da célula dificilmente é afetada no passo que a tecnologia avança, a carga crítica de uma célula de bit permanece aproximadamente constante (NICOLAIDIS, 2010, p. 34).

A vantagem de introduzir DRAMs com capacitância 3D foi que o volume da junção vulnerável é muito menor, o que resulta em um SER mais baixo por bit. Com o avanço da tecnologia, esse volume de junção está diminuindo ainda mais, entretanto o circuito de endereço na periferia da memória está se tornando mais suscetível a *soft errors* devido as capacitâncias de nó estarem diminuindo na mesma proporção (NICOLAIDIS, 2010, p. 35).

De acordo com Ziegler (1998) a taxa de erro nas DRAMs por células de memória depende fortemente do tipo de tecnologia que é aplicada para cada uma delas. Eles investigaram o SER de DRAMs em diferentes tipos de tecnologias, sendo elas do tipo *stacked-capacitor cells* (SC), *trench-cells* com carga externa (TEC) e *trench-cells* com carga interna (TIC). O SER de radiação cósmica foi realizado em 26 chips diferentes, cada um com 16 Mbits de DRAM. O SER das células TEC parece ser cerca de 1500 vezes maior que SER das células TIC, sendo

que as células SC possuem um SER intermediário, conforme ilustra a Figura 8.

Figura 8 – Efeito da radiação em diferentes tipos de tecnologias de DRAM: TEC (cima), SC (meio), and TIC (baixo)

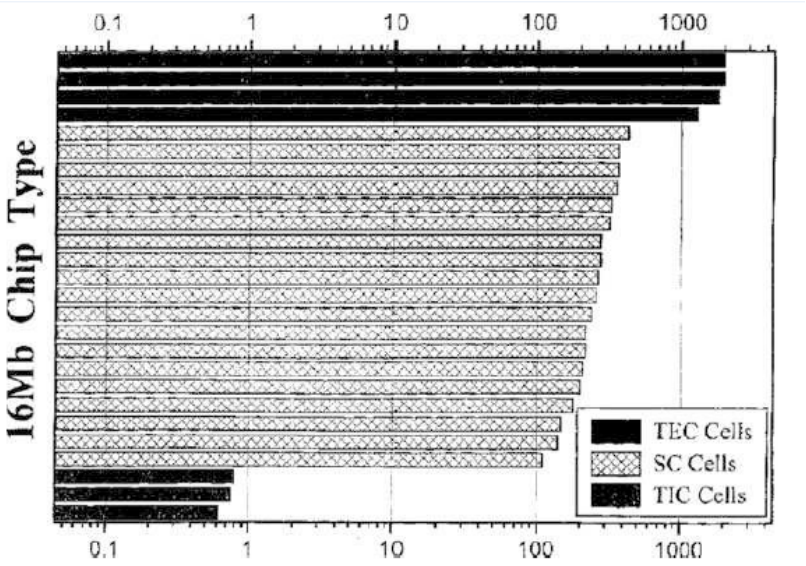


Figura retirada de Ziegler (1998)

2.7 PRINCÍPIOS DE TOLERÂNCIA A FALHAS

Existem muitos sistemas computacionais desenvolvidos para áreas de aplicações críticas, que precisam de um grau de confiança mais elevado, tais como: aplicações espaciais, controle de usinas de energia, telecomunicação, sistemas bancários, entre outros. Para esses sistemas, um mau funcionamento pode trazer grandes prejuízos, sendo que muitos deles não contam com uma fácil manutenção, tem instalações em locais de difícil acesso ou indisponível para isso. Assim, deve-se ter em mente que tais sistemas precisam ter técnicas e metodologias que previnam ou tolerem possíveis falhas, garantido o correto funcionamento ao longo do tempo.

Com o constante escalonamento tecnológico tem-se um aumento da sensibilidade de circuitos integrado a alguns efeitos causados pela radiação, como os SEEs. Muitas técnicas vêm sendo desenvolvidas para evitar a ocorrência de falhas devido a esse tipo de efeito, visto que com o aumento gradual na complexidade das novas arquiteturas, aumenta a dificuldade de desenvolver um sistema de alta confiabilidade.

Para sistemas aeroespaciais e aviônicos, existe um forte tendência em utilizar componentes de baixo custo disponíveis no mercado, os chamados *Components Off-The-Shelf* (COST), esses componentes agilizam o desenvolvimento de dispositivos tolerantes à radiação além de serem mais baratos e de fácil acesso (CABRAL, 2010). Para estes componentes, a ação da radiação é ainda mais presente causando grandes complicações em seus funcionamentos.

Para Coelho (2010) o conceito de confiabilidade se caracteriza pela capacidade de um sistema operar corretamente dentro de condições definidas, por um certo período de funcionamento. O conceito de disponibilidade se caracteriza na probabilidade de um sistema estar operacional quando a utilização deste for necessária.

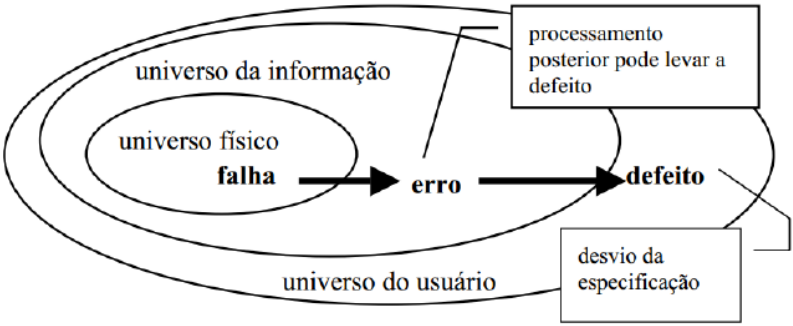
Para se adquirir a confiabilidade em um sistema de computação, na abordagem tradicional se busca utilizar tanto a prevenção quanto a tolerância de falhas. A prevenção de falhas diz respeito a reduzir todas as possíveis causas dos defeitos antes que o sistema entre em operação. A tolerância a falhas age a partir das possíveis falhas que podem surgir durante a vida útil do sistema. Três conceitos básicos estão relacionados à prevenção e tolerância a falhas: falha, erro e defeito.

2.8 FALHA, ERRO E DEFEITO

Uma falha (*fault*) pode ocorrer tanto em nível de hardware quanto de software, sendo esta a causa dos erro. No hardware pode ocorrer um defeito físico, uma imperfeição, ou uma alteração decorrente a fontes externas (como radiação, temperatura, entre outros), ou no software como o erro de lógica em determinado trecho de código do programa. Assim as falhas estão associadas ao universo físico ou ao projeto do sistema. O erro (*error*) e a representação da falha no universo da informação, é gerado quando, por consequência de uma falha, a informação é alterada. Por exemplo, a inversão de bit em uma célula da memória mudando o valor de uma variável. O defeito (*failure*) é perceptível ao usuário, quando a informação incorreta é processada sem que o erro seja detectado e tratado, levando ao usuário uma resposta errada do

sistema ou o não cumprimento de alguma tarefa. A Figura 9 representa os conceitos de falha, erro e defeito na forma de diagrama.

Figura 9 – Falha, Erro e Defeito



Fonte: Kruger (2014)

2.9 TOLERÂNCIA A FALHAS

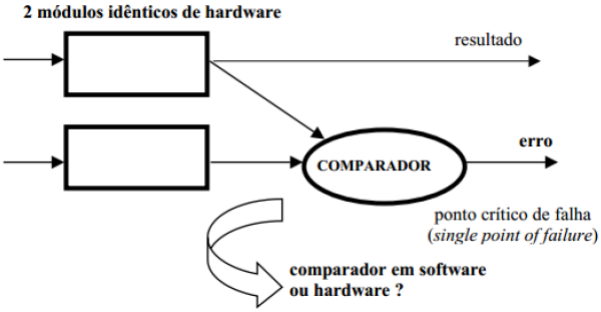
O princípio básico na construção de sistemas tolerantes a falhas é prover recursos extras (redundância) para superar os efeitos de um mau funcionamento e aumentar a segurança (*dependability*) do sistema. Alguns autores utilizam o termo "dependabilidade" em tradução da palavra *dependability*, porém esta palavra não existe na língua portuguesa, e será traduzido como segurança neste trabalho. Sistemas críticos são baseados, de um modo geral, em prevenção e tolerância a falhas e precisam garantir, entre outros requisitos, confiabilidade e disponibilidade. A estratégia mais comum em sistemas tolerantes a falhas é o uso de redundância em diferentes níveis de software e/ou hardware.

O uso de recursos redundantes empregados nesses sistemas geralmente não proporcionam uma melhoria de desempenho nem aproveitamento de espaço físico do dispositivo, pois seu objetivo é mascarar e contornar eventuais falhas, evitando o defeito. A redundância em sistemas tolerantes a falhas pode ser classificada em: hardware, software, e temporal (TAMBARA, 2014).

2.9.1 Redundância de hardware

A redundância de hardware consiste do uso de circuitos adicionais para detectar, e se possível, corrigir erros. As técnicas de detecção consistem, muitas vezes, em duplicação de componentes, onde a saída das cópias é enviada a um circuito comparador que identifica se existe ou não uma falha na resposta. Essas técnicas são chamadas de *Duplication With Comparison* (DWC), e não conseguem corrigir a falha, sem saber em qual componente ocorreu a falha. Um exemplo para esse tipo de técnica é exemplificado na Figura 10.

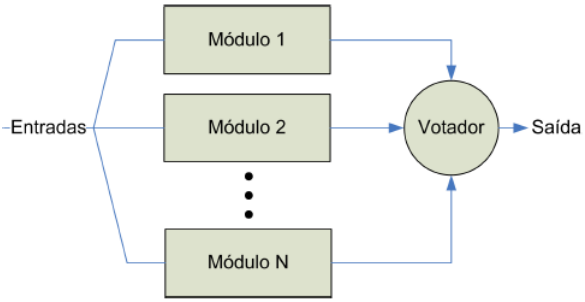
Figura 10 – Duplicação com comparação (DWC)



Retirado de Kruger (2014)

Um exemplo de redundância de hardware onde é possível detectar e corrigir uma falha é a redundância modular NMR (*N-uple Modular Redundancy*), mostrada na Figura 11. Nesse caso, existem $(2n + 1)$ módulos redundantes e, pelo menos $(n + 1)$ módulos são requeridos para concordar em um resultado, assim, esta estrutura é capaz de tolerar falhas em n módulos simultaneamente.

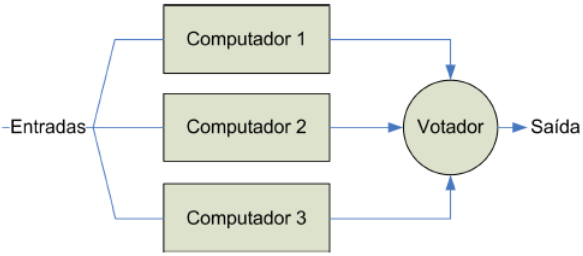
Figura 11 – Sistema NMR



Retirado de Coelho (2010)

A técnica mais utilizada em redundância de hardware é a Redundância Modular Tripla (*TMR - Triple Modular Redundancy*). Ela é uma versão da NMR para seu valor de $n = 1$, nesse tipo de implementação o sistema de votação costuma ser o ponto crítico no projeto, pois falhas no votador podem afetar a confiabilidade do sistema. Devido a essa fragilidade, implementações TMR em sistemas de votação têm sido as alternativas mais eficientes (COELHO, 2010). A Figura 12 ilustra a implementação de uma TMR.

Figura 12 – Sistema TMR



Retirado de Coelho (2010)

3 SISTEMA OPERACIONAL EMBARCADO

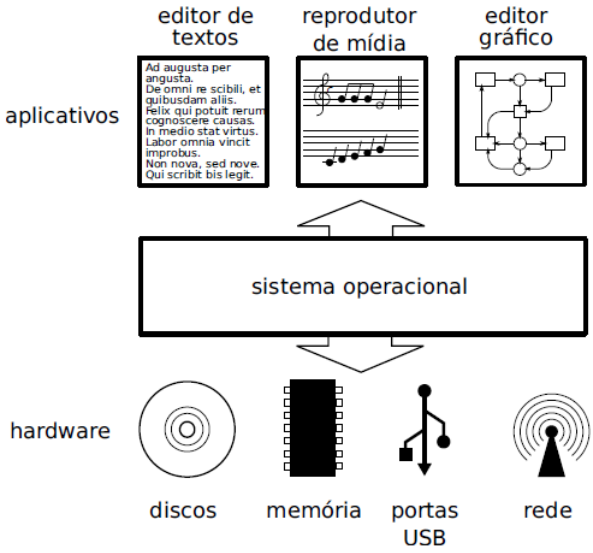
Neste capítulo serão apresentados alguns conceitos básicos de sistemas operacionais, características, tipos e aplicações. O capítulo também discorre sobre os sistemas operacionais embarcados (*embedded system*) e os sistemas operacionais de tempo real (*real time system*).

3.1 SISTEMAS OPERACIONAIS

O sistema operacional (SO) é a camada de software que opera entre os aplicativos de usuário final e o hardware. Ele abstrai a complexidade dos dispositivos e circuitos eletrônicos de baixo nível, tornando o desenvolvimento de aplicações de alto nível mais simples. Os circuitos são acessados através de interfaces, muitas vezes de instruções abstratas, onde as características e comportamentos mudam de acordo com a tecnologia usada na sua construção, por exemplo, a forma de acesso de um disco rígido e leitor de CD são diferentes, mesmo tendo o mesmo propósito. Assim, torna-se desejável fornecer um ambiente que implemente grande parte das funções comuns necessárias para os programas aplicativos, tornando o acesso homogêneo aos dispositivos físicos, permitindo a abstração das diferentes tecnologias de cada máquina.

Para Tanenbaum (2003), o sistema operacional consiste em uma camada de software que oculta o hardware e fornece ao programador um conjunto de instruções mais adequado. A Figura 13 descreve a arquitetura típica de um sistema de computação, contendo o sistema operacional, elementos de hardware e alguns programas aplicativos.

Figura 13 – Arquitetura geral de um sistema de computação



Extraído de: Maziero (2017)

Existem dois conceitos básicos que caracterizam os principais aspectos de um sistema operacional: gerência de recursos e abstração de recursos.

3.1.1 Gerência de recursos

Para um sistema com várias atividades simultâneas, na busca por completar seus objetivos, os programas aplicativos procuram utilizar ao máximo o hardware disponível, podendo assim surgir conflitos quando dois ou mais desses aplicativos precisam dos mesmos recursos para sua execução. Assim, o sistema operacional deve determinar políticas de uso dos recursos disponíveis no hardware, solucionando eventuais conflitos ou disputas. Um exemplo prático é uma impressora, cujo recurso deve ser usado por um aplicativo por vez (mutuamente exclusiva) para não ocorrer mistura de conteúdo nos documentos impressos, assim o SO define uma fila (FIFO - *First-In, First-Out*) de trabalhos a imprimir,

normalmente atendidos de forma sequencial.

3.1.2 Abstração de recursos

A tarefa de acessar um determinado recurso de hardware pode trazer grandes complicações devido às características específicas presente em cada dispositivo físico, assim o sistema operacional deve estabelecer interfaces de acesso aos dispositivos mais simples, diferente das interfaces de baixo nível. O SO deve também tornar os aplicativos independentes do hardware, tornando as interfaces de acesso homogênea para dispositivos com tecnologias distintas, permitindo assim que os aplicativos usem a mesma interface para dispositivos variados.

3.1.3 Tipos de Sistemas Operacionais

Os sistemas operacionais podem ser classificados de diversas maneiras: quanto ao tamanho, velocidade, suporte a recurso específico, acesso à rede, etc. Pode-se classificar a maioria dos SOs de acordo com as categorias abaixo:

- *Desktop:* é voltado ao usuário doméstico e empresarial para realizar atividades corriqueiras. São sistemas de tempo compartilhado, é construído para fazer a divisão justa dos recursos entre os processos. O tempo do processador é dividido em pequenas fatias de tempo, e o escalonador oferece ao processo uma fatia de tempo para usar o recurso. Esse sistema pode oferecer diferentes prioridades aos processos, implementado através de filas de prioridades, onde o escalonador vai atender primeiro os que estão presentes na fila de maior prioridade.

- *Servidor:* permite a gestão eficiente de grandes quantidades de recursos, como memória e processamento, possui suporte à operações em rede e oferece aplicações para o uso local em outros computadores da rede. Contém recursos multiusuário que impõem regras de controle de acesso e recursos a usuários não autorizados.

- *Distribuído:* possibilita que o usuário interaja com o sistema sem o conhecimento de onde está sendo executado ou armazenado seus dados, os recursos estão disponíveis em várias máquinas distribuídas globalmente.

- *Embarcado:* o sistema operacional embarcado é construído para operar sobre um hardware com poucos recursos de processamento, armazenamento e energia. Normalmente é desenvolvido como uma biblioteca para ser ligado ao programa da aplicação.

- *Tempo real:* sua característica principal é ter comportamento temporal previsível, seu tempo de resposta deve ser previsto no melhor e pior caso de operação. Sua estrutura é desenvolvida para minimizar esperas e latências imprevisíveis, sem tempos de acesso e sincronização excessivos.

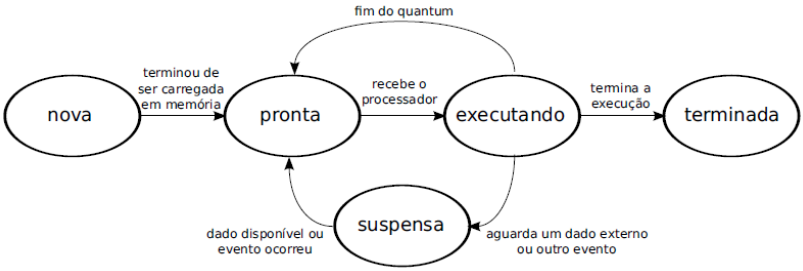
3.1.4 Tarefas

Tarefa ou atividade (*task*) é definida como sendo um fluxo sequencial de instruções executado pelo processador. Cada tarefa possui um estado interno bem definido, contendo valores de variáveis internas e posição atual de execução, elas interagem com outras entidades como usuários, periféricos e outras tarefas. Podem ser implementadas de várias formas, como processos ou *threads*.

Nos anos 40, os computadores executavam apenas uma tarefa de cada vez, eram sistemas primitivos, usados principalmente para aplicações de cálculo numérico com fins militares. Com o passar dos anos foi desenvolvido processadores que conseguem suspender a execução de uma tarefa, que esteja esperando dados externos por exemplo, e passar a executar outra tarefa. Assim passaram a suportar operações com múltiplas tarefas, chamado de sistema multitarefas. Nesse sistema, o processador deve executar todas as tarefas criadas pelo usuário, sendo que essas geralmente têm comportamentos, duração e importâncias distintas. Então o SO irá organizá-las de forma a decidir em que ordem serão executadas e com qual prioridade.

O ato de retirar uma tarefa e executar outra é denominado preempção, sistemas preemptivos geralmente utilizam o conceito de tempo compartilhado ou CTSS – *Compatible Time-Sharing System* (CORBATO, 1963) como forma de mensurar quanto cada tarefa pode usar o processador. Nesse sistema, cada tarefa recebe um limite de tempo de processamento, denominado quantum. Quando esgotado o quantum, a tarefa em execução deixa o processador e volta para uma fila de tarefas, aguardando sua oportunidade de executar novamente. O diagrama de estados de tarefas com preempção por tempo compartilhado é ilustrado na Figura 14. Seus respectivos estados serão explicados a seguir.

Figura 14 – Diagrama de estados de uma tarefa num sistema com preempção.



Extraído de: Kruger (2014)

- *Nova:* a tarefa está sendo criada, são executadas instruções junto com as bibliotecas necessárias, os dados estão sendo atualizados para permitir sua execução.
- *Pronta:* a tarefa está pronta para ser executada e fica aguardando sua vez numa fila, cuja ordem é determinada por um algoritmo de escalonamento.
- *Executando*: a tarefa está sendo executada pelo SO, junto com suas instruções e fazendo avançar seu estado.
- *Suspensa*: a tarefa não pode executar por estar aguardando a ocorrência de algum evento externo.
- *Terminada*: a tarefa tem seu processamento encerrado e tem seus recursos removidos da memória do sistema.

3.1.4.1 Contexto

Como foi dito anteriormente, a tarefa possui um estado bem definido, esse representa sua situação atual no programa, contendo a posição de código que está executando, valores de variáveis, entre outros. Esse estado é denominado contexto, e se modifica de acordo com a execução da tarefa. Nele são armazenados o valor do contador de programa (PC - *Program Counter*), do apontador de pilha (SP - *Stack Pointer*), e alguns registradores. Desse modo, cada contexto armazena

o estado interno do processador, incluindo informações sobre recursos utilizados pela tarefa.

3.1.4.2 Troca de Contexto

Quando o processador interrompe a execução de uma tarefa para executar outra, é preciso fazer operações para salvar e restaurar seu contexto sem corromper o estado interno da tarefa. Para isso, é salvo os valores de contexto atual em uma estrutura de dados no núcleo do SO. Essa estrutura é chamada de TCB (*Task Control Block*) ou PCB (*Process Control Block*). Além do contexto, são armazenados dados relacionados a gerência das tarefas, assim como: identificador da tarefa; estado da tarefa; área de memória usada; lista de recursos usados pela tarefa; prioridade, tempo de processamento já decorrido, volume de dados lidos/escritos, etc.

O armazenamento e recuperação do contexto e a atualização das informações contidas na TCB de cada tarefa são providos por um conjunto de rotinas realizada pelo despachante (do inglês *dispatcher*). E a escolha da próxima tarefa a receber o processador a cada troca de contexto é feita pelo escalonador (do inglês *scheduler*), essa escolha pode ser configurada para respeitar diversos fatores, como as prioridades, os tempos de vida e os tempos de processamento restante de cada tarefa.

3.1.5 Escalonamento de Tarefas

O escalonador define parte do comportamento do SO, ele permite que o sistema trate de forma mais eficiente e rápida as tarefas a serem executadas. Existem vários critérios que ajudam a estabelecer qual algoritmo de escalonamento utilizar, dependendo dos tipos de tarefas contidas no sistema. Podem ser levados em consideração os tempos de execução de cada tarefa (*turnaround time*), o tempo de espera na fila de prontas (*waiting time*), tempo de resposta (*response time*) de um evento ao sistema e outros como distribuição justa de tempo de processamento ou priorizando eficiência.

O escalonador de um SO pode ser preemptivo ou cooperativo. No sistema cooperativo, a tarefa executa permanentemente até seu término, podendo solicitar a liberação do processador, voltando para a fila de tarefas prontas. Em sistemas preemptivos, a troca de tarefas pode ocorre por vários motivos: quando seu *quantum* de tempo acaba,

quando executa uma chamada de sistema ou caso ocorra uma interrupção que ative uma tarefa de maior prioridade. A seguir são descritos os algoritmos de escalonamento mais usados em SOs.

3.1.5.1 Escalonamento FCFS

O algoritmo de escalonamento FCFS (do inglês, *First-Come First Served*) é o mais simples de implementar. Ele consiste em atender as tarefas na sequência de chegada na fila de prontas. Pode ser implementado de forma cooperativa, atendendo cada tarefa até seu término, ou de forma preemptiva, oferecendo uma fatia de tempo igual para cada tarefa na fila. Esse último é mais conhecido como escalonamento por revezamento, ou *Round-Robin*.

3.1.5.2 Escalonamento SJF

O escalonamento SJF (do inglês, *Shortest Job First*) consiste em atender a tarefa mais curta presente na fila de tarefas prontas. Ele proporciona os menores tempos médios de execução e de espera, seu maior problema consiste em estimar, a priori, a duração de cada tarefa antes de sua execução. Pode ser implementado como cooperativo, ou como preemptivo. Existe uma complexidade maior na versão preemptiva, visto que o escalonador deve comparar a duração prevista de cada nova tarefa que ingressa no sistema com o tempo restante de processamento das demais tarefas presentes, inclusive aquela que está executando no momento.

3.1.5.3 Escalonamento por prioridades

No escalonamento por prioridades, cada tarefa é associada a um valor de prioridade, esse valor é usado para escolher a próxima tarefa a receber o processador. Se for preemptivo, quando uma tarefa de maior prioridade se torna disponível, o escalonador entrega o processador a ela, deixando a atual na fila de prontas novamente. Para tarefas de mesma prioridades, o escalonamento pode ser implementado de várias formas para escolher a próxima tarefa a ser executada, como algoritmos de FCFS ou SJF.

3.2 SISTEMAS EMBARCADOS

A maioria das partes que compõem um satélite são embarcadas, como por exemplo o CDH e o EPS. Um sistema embarcado, ou sistema embutido, é qualquer sistema controlado por uma CPU (Unidade Central de Processamento), dedicado a um propósito particular, realizando um conjunto de tarefas pré-definidas com requisitos específicos. Geralmente sistemas embarcados são independentes, realizando seu trabalho com pouca ou nenhuma interferência do usuário. O papel do usuário é apenas gerar um evento, no qual se inicia uma sequência de atividades no sistema.

Um sistema embarcado é inserido em um ambiente para monitorar informações ou controlar dispositivos de interesses do usuário. Essas informações são obtidas através de diferentes tipos de sensores (como pressão, umidade, temperatura, etc), e controle por meio de atuadores como motor, válvulas e solenóides. Geralmente os sistemas embarcados devem transmitir esses dados para outros sistemas, isso ocorre através de interfaces e protocolos de comunicação como USB, BlueTooth, Rádio, ZigBee, entre outros.

Existem duas formas mais utilizadas para implementar um sistema embarcado. A primeira, sem o uso de um sistema operacional, usando apenas o loop principal para executar todas as tarefas. Essa abordagem de implementação é conhecida como *Super Loop* ou *Endless Loop* (PONT, 2002), e geralmente é dividida em duas partes: inicialização e loop infinito. Não requer suporte de tempo de execução e todas as inicializações (hardware, configurações de software e gerenciamento dos dispositivos) deverão ser incluídas no código de aplicação. Nesse sistema, todas as tarefas são executadas sequencialmente, na mesma ordem e até seu fim (sem preempção), e isso é um problema para estruturas mais complexas. A segunda forma de implementar um sistema embarcado é utilizando um sistema operacional de tempo real (RTOS - *Real-Time Operating System*), que será explanado com mais detalhes na próxima seção.

3.3 SISTEMAS OPERACIONAIS DE TEMPO REAL

Um RTOS é desenvolvido para atender requisitos temporais da aplicação, como por exemplo um simples relógio cronômetro ou uma aplicação mais complexa, como um sistema de navegação de um satélite. As principais funções de um RTOS são escalonar a execução das tarefas

em um tempo conhecido, gerenciar recursos do sistema e fornecer uma plataforma para o desenvolvimento de aplicações (LI e YAO, 2003).

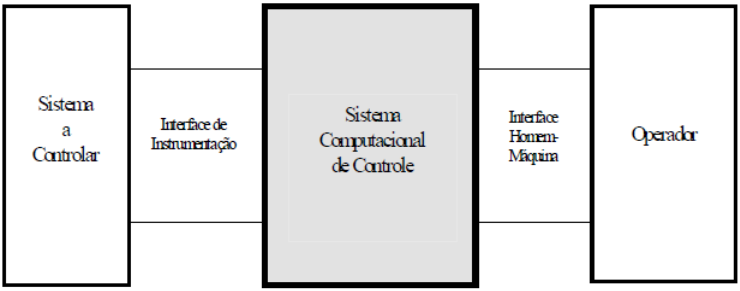
3.3.1 Sistemas de tempo real

Uma grande parte dos sistemas computacionais atuais interagem permanentemente com seus ambientes, dentre esses se destaca os Sistemas Reativos. Ele se caracteriza por reagir continuamente enviando respostas à estímulos de entrada vindos de seu ambiente. Um sistema de tempo real pode ser definido como um sistema reativo. Muitas vezes o termo "tempo real" é utilizado de forma inadequada, ao contrário da concepção usual, sistemas de tempo real não precisam ser necessariamente rápidos. A maior parte das aplicações de tempo real se comportam como sistemas reativos com restrições temporais.

Para Laplante (1992), sistemas de tempo real são aqueles que devem satisfazer precisamente restrições de tempo de resposta. Podendo causar falhas caso suas restrições não sejam atendidas.

De acordo com a IEEE (*Institute of Electrical and Electronics Engineers*), sistemas de tempo real são aqueles onde os resultados da computação podem ser usados para controlar, monitorar e responder a um evento externo em tempo (THOMAS; RANJITH; PILLAY, 2017). Assim computação de tempo real não quer dizer execução rápida, mas sim cumprindo prazos e gerando respostas suficientemente velozes para suprir as necessidades do processo. O comportamento correto desses sistemas diz respeito não apenas a integridade dos resultados gerados (correção lógica ou *"correctness"*), mas também aos valores de tempo em que são produzidos (correção temporal ou *"timeliness"*). Dessa maneira, a concepção do sistema de tempo real é diretamente relacionada ao seu ambiente de trabalho. A Figura 15 ilustra uma representação de um sistema de tempo real, contendo o sistema a controlar, o sistema computacional de controle e o operador.

Figura 15 – Representação de um Sistema de Tempo Real



Extraído de: Farines e Oliveira (2000)

3.3.2 Previsibilidade

O termo previsível (*"predictable"*), é uma característica de sistemas de tempo real no que diz respeito ao quanto o comportamento do sistema pode ser antecipado. Quando ocorrem variações à nível de hardware de cargas e de falhas, o sistema deve estabelecer, em hipótese, a máxima flutuação dessas variáveis. Existem duas hipóteses a ser consideradas nesse sistema:

- *Hipótese de carga:* mensura a carga computacional de pico (carga máxima) consebido pelo ambiente em um intervalo de tempo. Assim como eventos que ocorrem ocasionalmente, resultante de situações críticas.

- *Hipótese de falhas:* estabelece as possíveis falhas que ocorrem em tempo de execução, e com qual frequência elas ocorrem no sistema. Mantendo os seus requisitos funcionais e temporais.

3.3.3 FREERTOS

O FreeRTOS é um RTOS pequeno, simples e de código aberto. Além de muito confiável, é amplamente usado na indústria de microeletrônica, sua característica forte é sua portabilidade, ele suporta oficialmente mais de 20 arquiteturas diferentes, dentre as mais famosas

estão a PIC, ARM, Zilog Z80 e PC. Tem seu código fonte escrito em C com partes em assembler, e seu sistema é dividido em quatro arquivos bases (task.c, queue.c, croutine.c e list.c)(SERVICES, 2017).

A seguir serão apresentadas algumas características importantes de implementação e funcionamento do sistema FreeRTOS, onde serão abordados aspectos sobre tarefas, escalonador, filas, gerência de memória e estouro de pilha.

3.3.3.1 Tarefas no FreeRTOS

O FreeRTOS permite que um número ilimitado de tarefas sejam criadas e executadas, desde que aja hardware e memória suficiente para isso. As tarefas podem ser criadas ou destruídas, usando a função *vTaskCreate()* e *vTaskDelete()*, respectivamente. É possível trocar sua prioridade (usando *uxTaskPriorityGet()* ou *vTaskPrioritySet()*) ou requisitar um *delay* (com a função *vTaskDelay()* ou *TaskDelayUntil()* por exemplo). Existem outras formas de manuseio das tarefas que serão exemplificado posteriormente. A biblioteca das tarefas é escrita em linguagem C, têm como parâmetro um ponteiro *void** e retorna *void*.

No FreeRTOS, cada tarefa pode estar em apenas dois estados, executando (*running*) e não executando (*not running*). Para processadores de um núcleo, apenas uma tarefa assume o estado *"running"*, e as demais tarefas podem assumir 3 subcategorias do estado *"not running"*: bloqueada (*blocked*), suspendida (*suspend*) e pronta (*ready*).

- *Bloqueada:* a tarefa pode ser bloqueada por dois tipos diferentes de eventos. O primeiro por evento temporal (*time-related*), onde a tarefa espera por um determinado tempo fixo, por exemplo, com o uso da função de *delay vTaskDelayUntil()* para esperar um tempo de 10 milisegundos. A segunda forma de bloqueio é por sincronização de eventos, quando o evento tem origem de uma outra tarefa ou interrupção, como por exemplo esperar por um dado de uma fila (*queue*). Os eventos sincronizados no FreeRTOS podem ser gerados também usando semáforo (*semaphores*) e mutex.

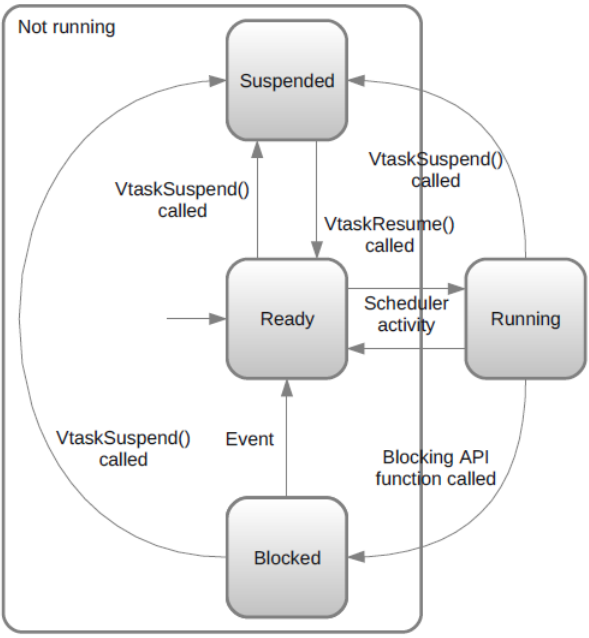
- *Suspendida:* tarefas no estado suspenso não estão disponíveis para o escalonador. O único caminho para o estado *"suspend"* é através da chamada de função *vTaskSuspend()*, e a saída é através das funções da *vTaskResume()* ou *xTaskResumeFromISR()*. A

maioria das aplicações não usam o estado suspenso.

- *Pronta:* As tarefas no estado *"ready"* ficam armazenadas na fila de prontas e esperam para sua execução.

O diagrama da Figura 16 apresenta uma simplificação dos estados das tarefas no FreeRTOS, incluindo as divisões do estado *"not running"*.

Figura 16 – Ciclo de vida das tarefas no FreeRTOS



Extraído de Services (2017)

3.3.3.2 Escalonamento de Tarefas no FreeRTOS

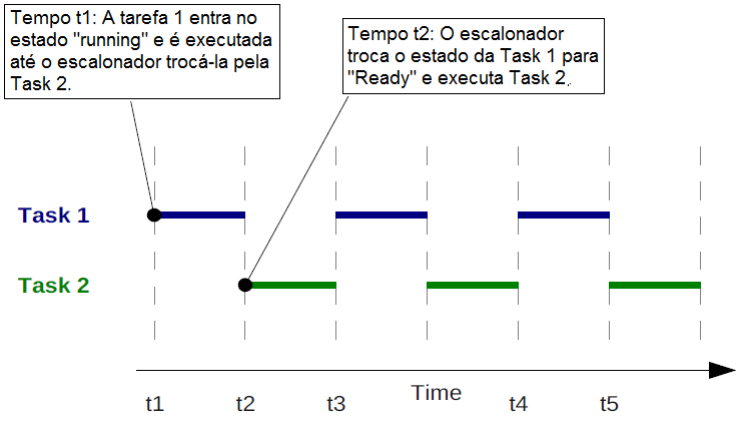
O escalonador escolhe qual tarefa no estado "*ready*" vai ser executada. A escolha é baseada apenas na prioridade das tarefas, para

cada tick de clock do processador, o escalonador decide qual tarefa deverá ser executada.

A prioridade é setada quando a tarefa é criada e seu valor é trocado apenas explicitamente pelo programador, o FreeRTOS não possui gerenciamento automático de prioridades. Quanto menor o valor, menor a prioridade, a prioridade mais baixa tem valor 0 (zero) e é reservada para a tarefa *idle*. O valor máximo da prioridade é definido na constante `MAX_PRIORITIES` (contido na biblioteca FreeRTOSConfig.h), onde a tarefa de mais alta prioridade é atribuída o valor de `MAX_PRIORITIES` -1.

Para tarefas de mesma prioridades, o escalonador distribui igualmente o processador. Quando duas tarefas estão no estado *"ready"* com prioridades iguais, a cada *quantum* o escalonador escolhe uma tarefa diferente para ser executada. O *quantum* é definido pelo tick de clock, onde seu valor é armazenado na constante `TICK_RATE_HZ`, em FreeRTOSConfig.h. Essa implementação é mais conhecida como *Round Robin* e é exemplificada na Figura 17.

Figura 17 – Duas tarefas de mesma prioridade no FreeRTOS



Adaptado de: Services (2017)

3.3.3.3 Gerenciamento de filas

As filas (*queues*) promovem um mecanismo de comunicação entre as tarefas (*task-to-task*) e entre as tarefas e interrupção (*task-to-interrupt*). Cada fila contém um tipo de dado fixo e um número máximo de itens, ambos são configurados quando a fila é criada. O número máximo de itens da fila é chamado de *"length"*. A fila é normalmente implementada como *buffer* FIFO (*First In First Out*), onde os dados são inseridos na cauda (*tail*) e removidos pela cabeça (*head*) da fila.

Existem dois métodos de se enviar dados numa fila, o primeiro é copiando o dado desejado para dentro da fila, criando uma fila especificamente do tipo de dado enviado; o segundo é copiando a referência do dado, tornando uma fila que armazena ponteiros. No primeiro, os dados enviados pela fila são copiados byte por byte, deixando a fila (em muitos casos) grande. No segundo, a fila carrega apenas os ponteiros para os dados, não os dados em si. O FreeRTOS recomenda que os dados armazenados na filas sejam passados via cópia (não passados por referência), isso apresentar algumas vantagens, como por exemplo:

- Poder enviar variáveis da *stack* diretamente pela fila, mesmo que a variável não exista mais na função onde foi declarada;
- A tarefa pode imediatamente reutilizar a variável enviada, mesmo que a outra tarefa não tenha lido a fila;
- A tarefas de envio não tem vínculos com as tarefas que recebem os dados. O desenvolvedor não precisa se preocupar com qual tarefa retém o dado e qual tarefa libera o dado;
- Num sistema com proteção de memória, a RAM que uma tarefa pode acessar é restrita. Se uma fila passa os dados por referência, a tarefa que envia e a que recebe poderão acessar a mesma área na qual os dados foram armazenados.

Se o tamanho dos dados armazenados na fila forem grandes, é preferível transferir seus respectivos ponteiros em vez de copiar o dado inteiro para dentro da fila. Fazer a transferência de ponteiros em filas é muitas vezes melhor escolha, por apresentar mais eficiência em requisitos de tempo de processamento e na quantidade de RAM necessária para sua criação. Entretanto, deve-se ter extremo cuidado no seu manuseio, a seguir será listado duas recomendações do FreeRTOS (BARRY, 2016) para uso de filas de ponteiros:

- É preciso garantir que as tarefas não modifiquem o conteúdo da memória ao mesmo tempo, invalidando o conteúdo do mesmo. Idealmente, somente a tarefa de envio deve ter permissão para acessar a memória até que o ponteiro tenha sido posto na fila. E somente a tarefa de recebimento deve ter permissão para acessar a memória após o ponteiro ter sido recebido da fila.

- Se a memória apontada foi alocada dinamicamente, então somente uma tarefa deve ser responsável pela liberação. Nenhuma tarefa deve tentar acessar a memória depois que ela for liberada. É recomendado também que um ponteiro nunca deve ser usado para acessar um dado alocado na pilha de tarefas. Os dados não serão mais válidos depois que o quadro da pilha tiver sido alterado.

3.3.3.4 Gerenciamento da memória *heap*

Para tornar o FreeRTOS mais fácil de ser usado, os objetos criados pelo kernel não são alocados estaticamente em tempo de compilação, mas sim alocado dinamicamente em tempo de execução. Esses objetos (tarefas, filas, semáforos, etc) são armazenados no segmento de memória destinada a *heap*. Isso simplifica a aplicação, facilita o planejamento e minimiza o espaço ocupado pela RAM.

A *heap* é declarada estaticamente em um *array*, onde ela é dividida em pequenos blocos de memória para fazer a alocação. O tamanho total (em bytes) do *array* é definido na variável *configTOTAL_HEAP_SIZE* em FreeRTOSConfig.h.

O FreeRTOS implementa funções de alocação de memórias diferente da biblioteca C padrão, isso por que a biblioteca padrão é desenvolvida para aplicações de uso geral, e nem sempre são adequadas para sistemas embarcados de tempo real. As funções desenvolvidas pelo FreeRTOS que substituem as funções *malloc()* e *free()* são *pvPortMalloc()* e *vPortFree()* respectivamente.

Pelo fato de existirem diferentes tipos de sistemas embarcados, com sua própria forma de usar a memória e com variados requisitos de tempo, o FreeRTOS criou uma nova camada de portabilidade para alocação de memória, deixando-a fora do seu código principal. Isso faz com que o usuário possa usar diferentes implementações da memória *heap*, podendo escrever suas próprias aplicações. Além disso, o FreeRTOS disponibiliza 5 diferentes tipos de alocação de memória: `heap_1.c, heap_2.c, heap_3.c, heap_4.c e heap_5.c`. A seguir

será explicado com mais detalhes cada uma delas.

- *Heap_1*: é a implementação mais simples do *pvPortMalloc()*, e não implementa a função *vPortFree()*. Foi desenvolvida para aplicações críticas que precisam de maior desempenho e segurança. É dedicada a pequenos sistemas embarcados onde o número de tarefas e objetos criados pelo kernel são bem definidos, assim a memória alocada continua a mesma durante todo tempo de vida da aplicação. Os sistemas críticos geralmente proíbem o uso de alocação dinâmica, devido às incertezas associadas ao determinismo, fragmentação de memória e alocações mal-sucedidas, a `heap_1` é sempre determinística e não fragmenta a memória.

- *Heap_2:* ela utiliza a função de liberação de memória *vPortFree()*, junto com o algoritmo de alocação de memória *"best fit"*. O algoritmo *best fit* busca o bloco de memória livre de tamanho mais próximo do requerido. A `heap_2` não combina os blocos livres adjacentes para criar um único bloco maior, gerando fragmentação. A `heap_2` é ideal para aplicações que criam tarefas de mesmo tamanhos, assim quando uma tarefa é excluída, o tamanho da *stack* alocada para uma nova tarefa vai ser o mesmo, diminuindo a fragmentação.

- *Heap_3:* usa as mesma funções *malloc()* e *free()* da biblioteca C padrão, o tamanha da *heap* é definido pela configuração do *linker*, não pelo FreeRTOS.

- *Heap_4:* usa como algoritmo de alocação de memoria o "*first fit*". Esse algoritmo busca o primeiro bloco de memória livre que tenha tamanho suficiente para alocar o que foi solicitado. Diferente da `heap_2`, a `heap_4` combina os blocos de memórias livres adjacentes, diminuindo o risco de fragmentação. A `heap_4` não é determinística, porém é mais rápida que a implementação de *malloc()* e *free()* da biblioteca C padrão.

- *Heap_5:* usa o mesmo algoritmo de alocação e liberação de memória da `heap_4`. Entretando, a `heap_5` não limita sua alocação em um espaço fixo na memória, ela pode alocar múltiplos e separados espaços de memória. A `heap_5` é usada quando a RAM provida pelo sistema não é um único bloco contínuo no mapa de memórias. É o único método de alocação de memória que deve ser explicitamente inicializado antes que seja chamado a função *pvPortMalloc()*. A `heap_5` é inicializada usando a função *vPortDe-*

fineHeapRegions(), e deve ser chamado antes que qualquer objeto do kernel seja criado.

3.3.3.5 Estouro da pilha

No FreeRTOS cada tarefa tem sua própria pilha (*stack*), assim quando a tarefa é criada usando a função ***xTaskCreate()***, sua pilha é alocada automaticamente na *heap*. A ocorrência do estouro da pilha (*stack overflow*) é um erro muito comum que causa instabilidade na aplicação. O FreeRTOS contém dois mecanismos que podem ser usados para detecção e *debug* do estouro da pilha. Sua configuração é realizada na constante `configCHECK_FOR_STACK_OVERFLOW`, e seus dois métodos são:

- Método 1: quando a constante `configCHECK_FOR_STACK_OVERFLOW` for configurada no valor igual a 1, o método de detecção de estouro de pilha verifica o contexto da tarefa durante sua troca, se o valor do ponteiro para a pilha for inválido (faz referência a uma área de memória fora da pilha), então há um erro e a função de estouro da pilha[1] é chamada.
- Método 2: quando a constante `configCHECK_FOR_STACK_OVERFLOW` for configurada no valor igual a 2, além de exercer a mesma detecção do método 1, o método 2 verifica se uma região da pilha não foi sobrescrita depois da troca de contexto. Ele verifica se os últimos n bytes da pilha permanecem inalterados (sobrescritos). A função de estouro da pilha é chamada caso esses bytes estiverem mudados. O método dois é menos eficiente que o método um, mas ainda é rápido.

[1] A função chamada no estouro da pilha é a `vApplicationStackOverflowHook()`

4 AMBIENTE DE AVALIAÇÃO DO SISTEMA OPERACIONAL FREERTOS

Neste capítulo será apresentada a arquitetura e as principais características do sistema de comando e tratamento de dados utilizado para avaliar o FreeRTOS.

4.1 ARQUITETURA DE HARDWARE DO CDH

A arquitetura do sistema será baseada no CDH do picossatélite Compass-1, desenvolvido na Universidade de Ciências Aplicadas de Aachen, na Alemanha (SCHOLZ; AACHEN, 2004). O Compass-1 foi projetado por dez graduandos de diferentes ramos da engenharia. Ele segue as especificações de CubeSat publicado pelas universidades Stanford e Calpoly (LEE, 2017). Tem uma estrutura cúbica de 10 cm de lado, peso abaixo de 1 kg e consumo energético de 1,5 W. A Figura 18 ilustra o Compass-1 com sua estrutura montada.

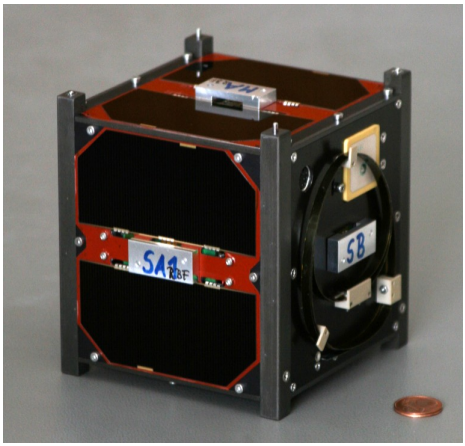


Figura 18 – Picossatélite Compass-1, extraído de: www.raumfahrt.fh-aachen.de/compass-1/home.html

O CDH do Compass-1 interliga todos os outros subsistemas por meio de uma conexão física, sendo essa elétrica e lógica. A Figura 19 ilustra essa interconexão. A comunicação e a entrega de energia entre o CDH e os subsistemas COM (*Communications System*), ADCS

(*Attitude Determination and Control System*), EPS (*Electrical Power System*) e TCS (*Thermal Control System*) é feita através do mesmo barramento.

Na alimentação elétrica estão disponíveis dois níveis de tensão de 3V e 5V, caso o sistema elétrico de potência entre em modo de economia, essas tensões são desligadas e apenas itens críticos são mantidos ativos, usando as linhas $3V_E$ e $5V_E$ que são permanentemente conectadas à energia.

Existe uma conexão dedicada ao P/L *(Payload)*, feita através de um cabo FCC (*Flexiva Compact Cable*) de 20 pinos. Essa quantidade alta de conexões dedicadas se dá pela complexidade de manuseio do dispositivo (câmera).

O número de interfaces foi dimensionado para ser o mínimo possível, quanto maior o número de interfaces, maior a complexidade do sistema, com isso aumentaria também as possíveis fontes de falha (PREISKER; SCHOLZ, 2004).

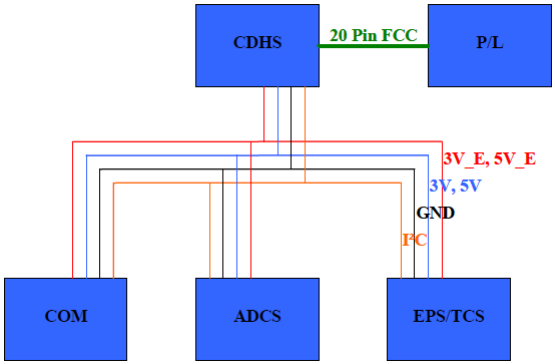


Figura 19 – Diagrama de interfaces do Compass-1, retirado de Scholz e Aachen (2004)

O Compass-1 foi dimensionado para ter custo baixo, onde não foi utilizado material propriamente espacial, que em sua maioria são mais caros. Isso aumenta em grande escala os riscos de mau funcionamento dos componentes, então o Compass-1 foi planejado da forma mais simples e robusta possível, reduzindo ao máximo sua complexidade e assim diminuindo suas chances de erro.

O CDH do Compass-1 é composto por três unidades, o microcon-

trolador (MCU) com seus periféricos, a unidade memória e a interface com o *Payload*. Essa estrutura é apresentada na Figura 20.

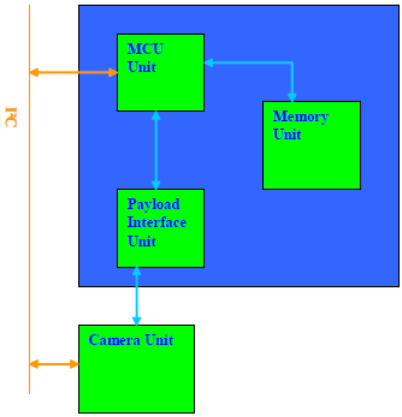


Figura 20 – Arquitetura do CDH do Compass-1, Scholz e Aachen (2004)

Os procedimentos iniciais de *boot* do CDH são acionados pelo EPS, onde são introduzidas as rotinas de verificações básicas e configurações necessárias para ativar o sistema. Em seguida, é passado para o loop principal, onde é executado a agenda de vôo. Essa agenda é atualizada por comandos recebido dos demais subsistemas, principalmente daqueles vindos da terra. Cada comando possui uma prioridade e cada tarefa um tempo a ser executada. As tarefas do CDH do Compass-1 são as seguintes:

- Parâmetros ADCS: comunicação entre o CDH e ADCS, para setar parâmetros iniciais de controle do subsistema;
- Transmitir dados: comunicação entre CDH e COM para transmitir dados dos sensores armazenados na memória para a terra;
- Transmitir imagem: comunicação entre CDH e COM para enviar imagens armazenadas na memória para a terra;
- Captura Imagem: comunicação entre CDH e Payload, envia o comando de captura de imagem para a câmera e armazena a imagens na memória;

- Captura de Dados: comunicação entre CDH e EPS/ADCS, armazena os dados de sensores e atuadores dos subsistemas.

4.1.1 Unidade microcontroladora

Para a unidade microcontroladora do CDH desenvolvido será utilizado o microcontrolador PIC24FJ128GA010 da Microchip (Figura 21), que é um dispositivo com arquitetura de 16-bits Harvard modificada, contendo 100 pinos, 128kB de memória de programa, 8KB SRAM, 5 Timers e 16 conversores A/D.

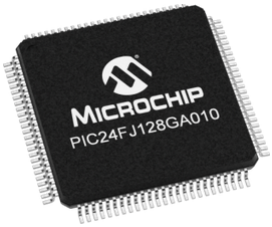


Figura 21 – PIC24FJ128GA010, Microchip

4.2 ARQUITETURA DE SOFTWARE DO CDH

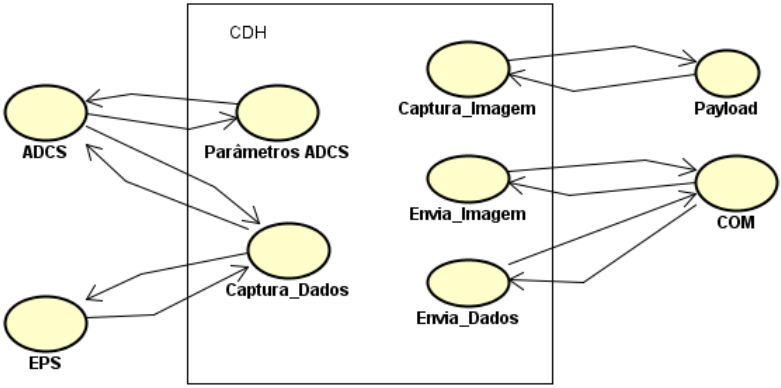
O CDH desenvolvido neste trabalho será implementado usando o FreeRTOS como sistema operacional. E nele será criado as cinco tarefas, mostradas anteriormente, executadas pelo picossatélite Compass-1. Será configurado também as mesmas prioridades, sendo que nenhuma das tarefas terão prioridades iguais. A Tabela 5 relaciona o nome, a prioridade e taxa de execução diária dessas tarefas.

Tabela 5 – Tarefas do CDH

Tarefa	Prioridade	Taxa
Parâmetros ADCS	5	1/Vida
Captura de Dados	4	1/Hora
Transmitir Dados	3	1/Hora
Captura Imagem	2	12/dia
Transmitir Imagem	1	12/dia

Cada tarefa criada permanecerá ativa até o fim da missão do satélite, sem ela ser destruída ou recriada. Ao término da execução de suas rotinas diárias, é acionado um *delay* que tira a tarefa de execução e deixa outra apta a ser executada. Esse *delay* é definido de acordo com a taxa de execução de cada tarefa. Essa taxa diz respeito à frequência que a tarefa irá ser acionada diariamente. A Figura 22 ilustra essas tarefas e com quais subsistemas elas se comunicam.

Figura 22 – Tarefas do CDH



4.2.1 Escalonador

O escalonador do FreeRTOS foi configurado para ser preemptivo baseado na prioridade. Cada tarefa será executada enquanto não houver uma de maior prioridade na fila de prontas, caso isso ocorra, ela será preemptada e a tarefa de maior prioridade deverá ser executada.

4.2.2 Memória Heap

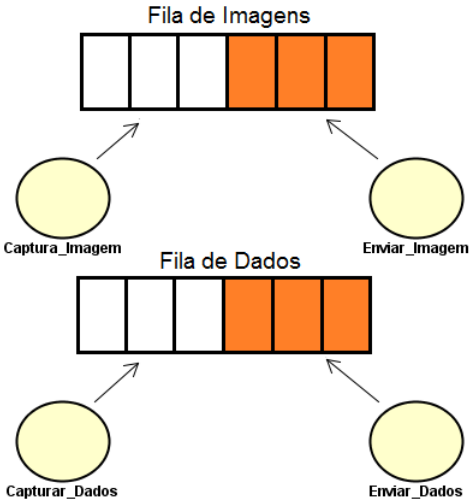
A implementação de memória *heap* escolhida para o desenvolvimento de testes no FreeRTOS foi `heap_1`, ela apresentar maior segurança e melhor desempenho que as demais bibliotecas desenvolvidas pelo FreeRTOS. Como existe um número fixo de tarefas geradas e essas nunca serão destruídas ou recriadas, não se faz necessário o uso da

função para liberação de memória (*free()*). A `heap_1` soluciona os problemas relacionados a fragmentação de memória e falta de determinismo.

4.2.3 Comunicação entre tarefas do CDH

A comunicação e troca de dados entre as tarefas é feita por filas. Foram criadas duas filas, uma entre as tarefas `Captura_Imagem` e `Enviar_Imagem`; e outra entre `Captura_Dados` e `Enviar_Dados`. As tarefas de captura, tem a função de receber dados dos outros subsistemas e disponibilizá-los para as tarefas de envio. Assim as tarefas de envio copiam os dados da fila e os transmitem para o subsistema COM, responsável por encaminhar esses dados para a terra. A Figura 23 ilustra essa troca de dados entre as tarefas de captura e envio.

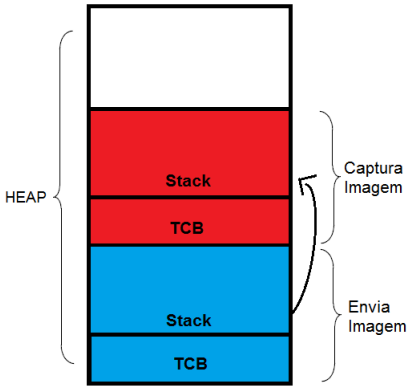
Figura 23 – Troca de dados entre as tarefas de captura e envio



Como as imagens capturadas pela câmera tem uma grande quantidade de bytes, é mais eficiente criar uma fila de ponteiros para o manuseio dessas imagens. Assim a tarefa de envio tem acesso direto ao local das imagens armazenadas pela tarefa de captura, sem ser preciso criar uma fila propriamente de imagens e sim de ponteiros para as

imagens. A Figura 24 ilustra esse compartilhamento de memória entre essas tarefas.

Figura 24 – Compartilhamento de memória



Para garantir que não ocorra os erros mencionados no capítulo anterior sobre fila de ponteiros (Seção 3.3.3.3), foi criando um mutex que garante o acesso de cada tarefa ao local de armazenamento, sem corromper o dado.

4.2.4 Confiabilidade

Para testar a confiabilidade do sistema FreeRTOS, deve-se assegurar que o sistema funcione mesmo em situações críticas, tendo como maior característica sua previsibilidade. Para isso, o sistema deve operar em situações que exijam sua carga máxima, num ambiente com pontos de falhas. Para tanto, foi necessário estipular hipótese de falha e hipótese de carga.

4.2.4.1 Hipótese de falha

Para se adquirir valores aproximados de falhas causadas pela radiação do tipo SEU, é preciso estipular em qual órbita e inclinação terrestre o nanossatélite irá estar, assim se sabe aproximadamente a quantidade de radiação que este satélite irá receber. De acordo com a

NASA (1996), veículos espaciais ou satélites em baixa inclinação (< 28 graus) estando na baixa órbita terrestre (LEO, < 500 km) em ambos os hemisférios (Norte e Sul) podem receber dose de radiação (TID) entre 100 a 1000 rad/ano. Para veículos espaciais ou satélites em alta inclinação ($28 < I < 85$ graus) na LEO em ambos os hemisférios, a taxa de TID típica é de 1000 a 10,000 rad/ano.

A Tabela 6 relaciona a dose total de radiação que tipicamente os diferentes tipos de componentes suportam, diferenciando daqueles produzidos comercialmente (*comercial*), produzidos com tolerância a radiação (*rad tolerant*) e produzidos com alta tolerância a radiação (*rad hard*). A Tabela 6 relaciona também a taxa de erro de SEU para os diferentes tipos de componentes.

Tabela 6 – Categoria de componentes produzidos comercialmente

Categoria	Dose total (típica)	Taxa de erro de SEU
Comercial	2 a 10 krad	10^{-5} erro/bit-dia
Rad Tolerant	20 a 50 krad	10^{-7} - 10^{-8} erro/bit-dia
Rad Hard	200 krad à 1 Mrad	10^{-10} - 10^{-12} erro/bit-dia

Valores retirados de: NASA (1996)

Se utilizado um microcontrolador produzido comercialmente (*comercial*) para o CDH com sua órbita na LEO em alta inclinação, baseado nos dados apresentados Tabela 6, sua vida útil duraria aproximadamente 2 meses e 12 dias, devido a quantidade total de radiação absorvida de 10.000 rad/ano (em seu pior cenário), considerando que este dispositivo suporta uma dose de até 2000 rad (pior cenário). Em sua memória RAM de 8 kB, ocorreria em média 0,64 erros por bit em um dia, devido a taxa de ocorrência de SEU de 10^{-5} erro/bit ao dia.

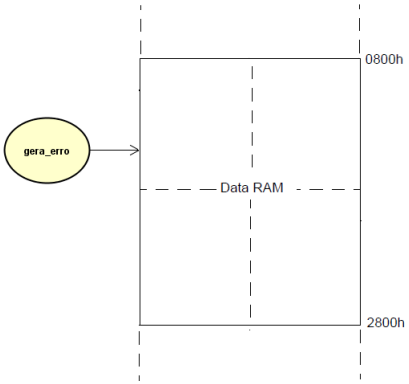
5 AVALIAÇÃO DO SISTEMA OPERACIONAL FREERTOS

Neste capítulo serão apresentados os resultados obtidos com os testes realizados para analisar a confiabilidade do FreeRTOS aplicado em unidades CDH em nanossatélites. Foram realizados 5 experimentos nesse sistema, variando a quantidade de carga, a quantidade de erros gerados nas diferentes partes da memória, formando assim um conjunto de informações que são utilizados na análise de confiabilidade do FreeRTOS para nanossatélites.

5.1 DESENVOLVIMENTO

A geração de erros aleatórios em memórias representando o tipo SEU foi feita a partir de uma tarefa criada no FreeRTOS descrita como `gera_erro`. Ela cria aleatoriamente uma troca de bit em qualquer parte da memória de dados do programa. A representação da tarefa `gera_erro` é ilustrada na Figura 25.

Figura 25 – Tarefa `gera_erro` na memória



A frequência na qual a tarefa `gera_erro` é acionada foi calculada de acordo com a Tabela 6, considerando que a quantidade de memória (M) RAM em bit, e uma taxa (Tx) de erro/bit-dia. O cálculo da frequência diária de erro (F) é realizado com base na Equação 5.1.

$$F = Tx.M \tag{5.1}$$

Para um microcontrolador produzido comercialmente sem tolerância à radiação, a taxa de erro/bit-dia é de 10^{-5}, com memória de 8KB, a frequência de erros por dia é igual a:

$$F = 10^{-5}.8000.8 = 0.64(erro/dia) \tag{5.2}$$

Ou seja, ocorre 0.64 erros por dia, aproximadamente um a cada 37 horas e 30 minutos.

Para que os testes possam ser feitos em um espaço de tempo menor foram acelerados os tempos de execução de cada tarefa e erros gerados. A Tabela 7 lista os valores da taxa real e a taxa para teste.

Tabela 7 – Taxa de execução das tarefas nos testes

Tarefa	Taxa para Teste	Taxa Real
Parâmetros ADCS	1/Vida	1/Vida
Captura de Dados	3ms	1/Hora
Transmitir Dados	3ms	1/Hora
Captura Imagem	6ms	12/dia
Transmitir Imagem	6ms	12/dia
Gera Erro	112ms	37h 30mim

Valores para a tarefa *Parâmetros ADCS* não foram modificados por ela ser executada apenas uma vez com prioridade máxima, garantido ser a primeira tarefa processada.

5.2 EXPERIMENTO 1

O experimento 1 consiste em avaliar quanto tempo o sistema continua funcionando sem erros em sua carga máxima. Nesse experimento não foram gerados erros. Ele serviu apenas para estimar a quantidade de carga que o sistema suporta sem apresentar erros. Para tanto, foram testados os níveis de carga que variam de 60% até 100% do máximo que o microcontrolador suporta. Os resultados obtidos estão listados na Tabela 8 .

Tabela 8 – Experimento 1 - hipótese de carga

Carga	Comportamento	Tempo
80%	sem erro	maior que 1 ano
90%	sem erro	maior que 1 ano
96%	sem erro	maior que 1 ano
98%	sem erro	maior que 1 ano
99%	erro[1]	0.144ms
100%	erro	0.144ms

5.3 EXPERIMENTO 2

O experimento 2 consiste em avaliar como o FreeRTOS se comporta mediante geração de falhas do tipo *sorf error* na memória. A memória RAM total do microcontrolador é de 8KB, onde a geração de erros se fez em toda ela. Considerando que o dispositivo é produzido sem qualquer tipo de tolerância à radiação, a média de erros produzidos diariamente é de 0,64 erros por bit. Os dados gerados neste experimento estão no APÊNDICE A.1, os resultados obtidos são:

- Testes: 10
- Tempo de duração média da execução sem apresentar defeitos: 38 dias e 3 horas.
- Duração máxima de execução sem apresentar defeitos: 57 dias e 19 horas.
- Duração mínima de execução sem apresentar defeitos: 20 dias e 7 horas.
- Média de erros geradas: 24.4 falhas.
- Defeito de maior ocorrência: StackOverflow com 80% dos casos.

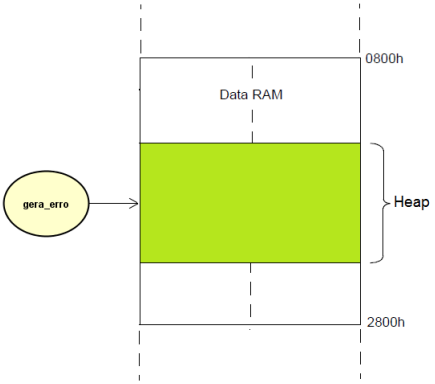
StackOverflow - Chamada de função *vApplicationStackOverflowHook()*.

[1] Falha de segmentação - inválido endereço de memória

5.4 EXPERIMENTO 3

Neste experimento foram gerados erros apenas na região de memória destinada a *heap*. Assim pode-se avaliar com mais clareza em qual parte do programa o sistema é mais suscetível a defeitos. A Figura 26 evidencia essa área de memória (região verde) na qual foram gerados os erros.

Figura 26 – Tarefa `gera_erro` na memória *heap*



Também foi considerado que o dispositivo é produzido sem qualquer tipo de tolerância à radiação. A média de erros gerada diariamente é de 0,64 erros por bit. Os dados gerados neste experimento estão no APÊNDICE A.2, os resultados obtidos são:

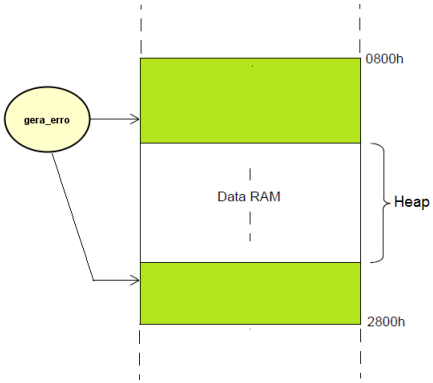
- Testes: 10
- Tempo de duração média da execução sem apresentar defeitos: 20 dias e 18 horas.
- Duração máxima de execução sem apresentar defeitos: 46 dias e 20 horas.
- Duração mínima de execução sem apresentar defeitos: 3 dias e 3 horas.
- Média de erros geradas: 13.3 falhas.

- Defeito com maior ocorrência: StackoOverflow com 60% dos casos.

5.5 EXPERIMENTO 4

O experimento 4 consiste em gerar erros na memória desconsiderando o espaço onde se localiza a *heap*. A Figura 27 evidencia essa área de memória (região verde) na qual foi gerado os erros.

Figura 27 – Tarefa `gera_erro` fora da memória *heap*



Também foi considerado que o dispositivo é produzido sem qualquer tipo de tolerância à radiação. A média de erros gerada diariamente é de 0,64 erros por bit. Os dados gerados neste experimento estão no APÊNDICE A.3, os resultados obtidos são:

- Testes: 10
- Tempo de duração média da execução sem apresentar defeitos: 46 dias e 6 horas.
- Duração máxima de execução sem apresentar defeitos: 93 dias e 18 horas.
- Duração mínima de execução sem apresentar defeitos: 17 dias e 4 horas.
- Média de erros geradas: 29.6 falhas.

- Defeito maior ocorrência:: `_defaultInterrupt()`[2] com 70% dos casos.

5.6 EXPERIMENTO 5

O experimento 5 visa avaliar como os dispositivos produzidos com tolerância a radiação (*rad tolerant*) se comportam mediante geração de erros do tipo SEU no FreeRTOS. De acordo com a Tabela 6, esses dispositivos contém uma taxa de erro de 10^{-7} erro/bit-dia (pior caso), que de acordo com a Equação 5.1 a frequência de erro é:

$$F = 10^{-7}.8000.8 = 0.0064(erro/dia) \tag{5.3}$$

Considerando que a memória RAM total do microcontrolador é de 8KB e a geração de erros foi realizada em toda ela. Os dados gerados neste experimento estão no APÊNDICE A.4, os resultados obtidos são:

- Testes: 10
- Tempo de duração média da execução sem apresentar defeitos: 163 dias e 10 horas.
- Duração máxima de execução sem apresentar defeitos: 384 dias e 2 horas.
- Duração mínima de execução sem apresentar defeitos: 26 dias e 1 horas.
- Média de erros geradas: 25.1 falhas.
- Defeito mais ocorrente: stackoverflow() com 60% dos casos.

5.7 DISCUSSÃO DOS RESULTADOS OBTIDOS

Em primeira análise, o FreeRTOS tem plenas condições de ser utilizado em missões espaciais de pouca duração (menor que 1 mês). Se utilizado com um microcontrolador de pouca memória (como o PIC24), baseado no tempo de duração média do Experimento 2, o FreeRTOS

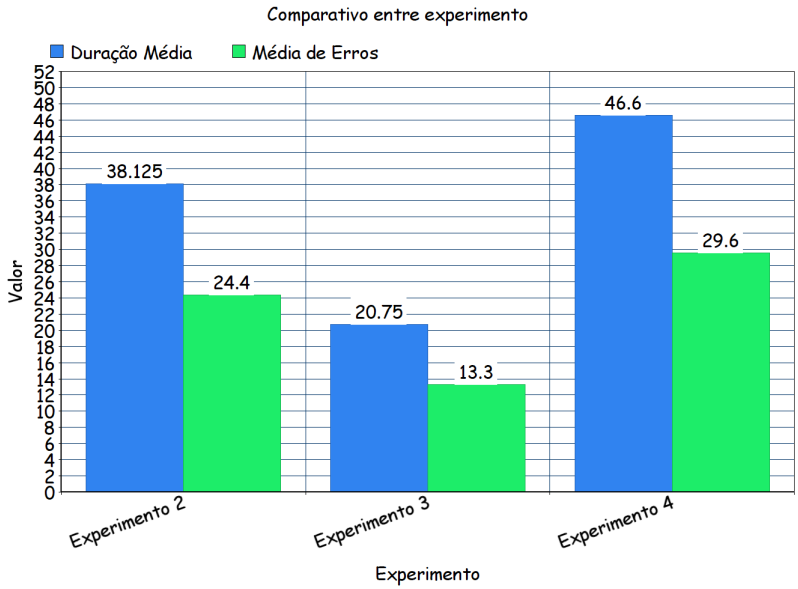
[2] `_defaultInterrupt()` é um ISR (*Interrupt Service Routine*) padrão gerado pelo compilador quando em modo de debug. Ele é acionado quando o programa tenta acessar uma parte da memória que não deveria.

pode funcionar um pouco mais de um mês no espaço sem apresentar defeitos.

No Experimento 1, o FreeRTOS se mostrou extremamente eficiente em relação a quantidade de carga suportada pelo mesmo. Assim o sistema não tem problemas em trabalhar com até 98% de sua carga máxima. Vale ressaltar que os experimentos restantes não foram feitos com carga elevada, visto que projetos de satélites e dispositivos críticos geralmente são projetados para serem o mais robusto possível, então os demais testes foram desenvolvidos com apenas a carga necessária para suas funções e tarefas descritas no capítulo anterior.

O gráfico ilustrado na Figura 28 apresenta os resultados dos experimentos 2, 3 e 4.

Figura 28 – Comparativo entre experimentos 2, 3 e 4



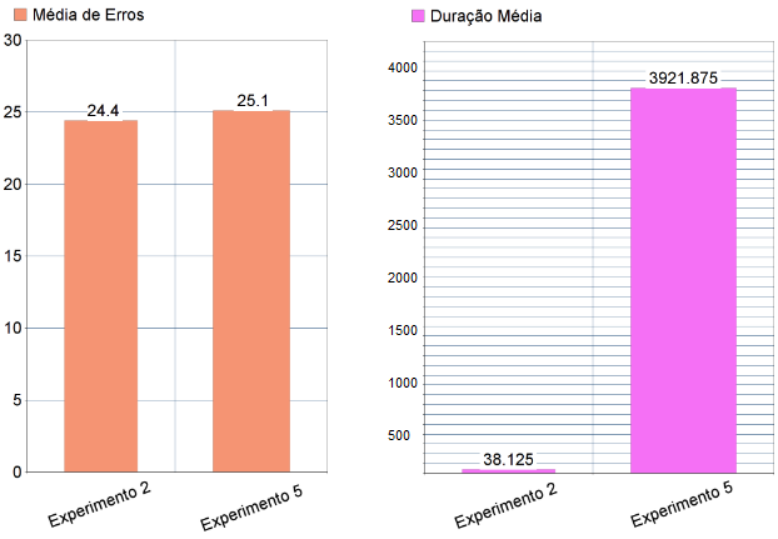
Pode se notar que o FreeRTOS tem uma fragilidade maior em sua memória *heap* quando nela é gerado erros do tipo SEU. Quando comparado às médias dos experimentos gerados apenas na memória *heap* e desconsiderando ela, tem se um tempo duas vezes maior de um em relação ao outro. Assim a área de memória destinada a *heap* é duas

vezes mais susceptível a erros que as demais áreas da memória.

Apesar do FreeRTOS ser mais vulnerável em sua memória *heap*, ele tem um grande controle da mesma. Isso se torna evidente quando comparado os tipos de erros testados nos Experimentos 3 e 4. No experimento 3, na maioria dos testes o FreeRTOS identificou o defeito na memória e efetuou a chamada de função `vApplicationStackOverflowHook()`, na qual impede a aplicação de continuar executando, devido a ocorrência de um erro na *stack* da tarefa. Enquanto que no Experimento 4 não houve chamada de função de estouro da pilha, no caso, a maior ocorrência de erro foi dado pelo *debugger*, pela chamada de função `_DefaultInterrupt( )`, na qual desativa todas as interrupções na ocorrência de um erro.

O gráfico ilustrado na Figura 29 compara os resultados obtidos com os experimentos 2 e 5.

Figura 29 – Comparativo entre o experimento 2 e 5



Os resultados do Experimento 5 foram muito próximos dos resultados obtidos com o Experimento 2, comparando com a média de erro e os erros gerados. A disparidade dos tempos de duração média entre os experimentos também eram esperados, visto que a taxa de erro do experimento 2 era 100 vezes maior que a do experimento 5, fazendo

com que a duração média aumentasse proporcionalmente a esse valor.

6 CONSIDERAÇÕES FINAIS E PROPOSTAS PARA TRABALHOS FUTUROS

Este trabalho simulou a ocorrência de falhas do tipo *Single Event Upset* em um sistema embarcado para analisar a confiabilidade do sistema operacional FreeRTOS em nanossatélites. O FreeRTOS foi desenvolvido para sistemas embarcados com diferentes aplicações e vem sendo grandemente utilizado em áreas comerciais críticas.

Os resultados deste trabalho apontaram um possível ponto de maior vulnerabilidade do sistema, a área de memória destinada a *heap*. Com os resultados apresentados, foi possível também mensurar a quantidade média de falhas do tipo SEU suportada pelo FreeRTOS, expondo também o tempo médio de duração de sua aplicação quando estas falhas eram geradas. Os resultados expressaram também a porcentagem de vezes que o FreeRTOS conseguiu identificar o erro na memória. Esses erros podem produzir resultados incorretos (defeitos) quando não são identificados. A presença de defeitos pode causar alguns transtornos, e em muitos casos, danos graves e irreversíveis. O uso do FreeRTOS é uma alternativa para nanossatélites no que tange confiabilidade e desempenho, onde se busca baixo custo e fácil desenvolvimento.

6.1 TRABALHOS FUTUROS

Nesta seção são listadas algumas propostas para trabalhos futuros que visam estender o trabalho desenvolvido nesta monografia de fim de curso.

1. Aplicar o uso das técnicas de tolerância a falhas nas partes vulneráveis do microcontrolador;
2. Estender o conjunto de testes em um ambiente real ou em ambiente de teste que reproduza a radiação ionizante;
3. Testar a confiabilidade de outros sistemas operacionais de tempo real;
4. Avaliar a precisão do *timer* em outras situações.

REFERĘNCIAS

BALEN, T. R. *Efeito da radiação em dispositivos analógicos programáveis (FPAAS) e técnicas de proteção.* [S.l.: s.n.], 2010.

BARRY, R. Mastering the freertos^{TM} real time kernel. 2016. <https://www.freertos.org/Documentation/161204$_{M}astering_{t}he_{F}reeRTOS_{R}ea$ $A_{H}ands - On_{T}utorial_{G}uide.pdf$>.

CABRAL, M. T. *Desenvolvimento de um protótipo de um sistema de determinação de atitude de satélites artificiais com tolerância a falhas.* [S.l.: s.n.], 2010.

COELHO, A. A. da P. *FT-OPENRISC 1200: um processador de arquitetura RISC tolerante a falhas para sistemas embarcados.* [S.l.: s.n.], 2010.

COHEN T.S. SRIRAM, N. L. D. M. S. B. N.; FLATLEY, R. Soft error considerations for deep-submicron cmos circuit applications. 1999.

FARINES, J. F. J.; OLIVEIRA, R. de. Sistemas de tempo real. 2000.

GARBER, S. *NASA History.* 2017. <http://history.nasa.gov/sputnik/>.

KRUGER, K. *Programação de Microcontroladores Utilizando Técnicas de Tolerância a Falhas.* [S.l.: s.n.], 2014.

LAPLANTE, P. A. *Real-Time Systems Design and Analysis: An Engineer's Handbook.* Piscataway, NJ, USA: IEEE Press, 1992. ISBN 0780304020.

LEE, S. *Cubesat Specification.* 2017. <http://cubesat.calpoly.edu/documents/cubesatspec.pdf>.

MACHADO, S. R. F. *Estudo de um processo de garantia da confiabilidade de sistemas eletrônicos embarcados a single event upsets causados por partículas ionizantes.* [S.l.: s.n.], 2014.

MARINAN, K. C. A. *From CubeSats to Constellations: Systems Design and Performance Analysis.* [S.l.: s.n.], 2013.

MAZIERO, C. A. Sistemas operacionais: Conceitos e mecanismos. 2017. <http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=so:so-livro.pdf>.

NASA. Space radiation effects on electronic components in low-earth orbit. 1996. <https://oce.jpl.nasa.gov/practices/1258.pdf>.

NICOLAIDIS, M. *Soft Errors in Modern Electronic Systems*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441969926, 9781441969927.

PREISKER, M. H. R.; SCHOLZ, A. Compass one, phase b documentation. 2004.

SCHOLZ, N. M. A.; AACHEN, F. Design and development of a cdhs for a pico satellite. 2004.

SEIFERT P. SLANKARD, M. K. B. N. V. Z. C. B. A. V. S. M. B. G. J. M. N. Radiation-induced soft error rates of advanced cmos bulk devices. Proc. Int'l Rel. Phys. Symp. (IRPS), p. 217–225, 2006.

SERVICES, A. W. The freertos^{TM} reference manual. 2017. <https://www.freertos.org/Documentation/FreeRTOS$_{R}eference_{M}anual_{V}$10.0.0.*pdf*>

TAMBARA, L. A. *Caracterização de circuitos programáveis e sistemas em chip sob radiação*. [S.l.: s.n.], 2014.

TANENBAUM, A. *Sistemas operacionais modernos*. Prentice-Hall do Brasil, 2003. ISBN 9788587918574. <https://books.google.com.br/books?id=meCAGQAACAAJ>.

THOMAS, J. V.; RANJITH, R.; PILLAY, R. V. Guaranteeing fault tolerance in real time systems under error bursts. In: *2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*. [S.l.: s.n.], 2017. p. 1480–1484.

TORRES, F. E. *Desenvolvimento de um Sistema de Emulação de Single Event Upsets em Dispositivos COTS Baseado na Metodologia Code Emulating Upsets*. [S.l.: s.n.], 2013.

ZIEGLER, M. E. N. J. D. S. R. J. P. C. J. G. H. P. M. C. J. M. J. F. Cosmic ray soft error rates of 16-mb dram memory chips. EEE J. Solid-State Circuits, vol. 33, no. 2, p. 246–252, 1998.

APÊNDICE A – Dados Obtidos com os Experimentos

A.1 TABELA DE RESULTADOS DO EXPERIMENTO 2

teste	Tempo de execução	Erro gerado	Erros gerados
1	21 dias 20 horas	StackOverflow	14
2	51 dias 13 horas	Tarefa 1 deixou de responder	33
3	56 dias 6 horas	StackOverflow	36
4	57 dias 19 horas	StackOverflow	37
5	40 dias 15 horas	StackOverflow	26
6	50 dias	Tarefa 2 deixou de responder	32
7	20 dias 7 horas	StackOverflow	13
8	29 dias 16 horas	StackOverflow	19
9	29 dias 2 horas	StackOverflow	19
10	23 dias 10 horas	StackOverflow	15

A.2 TABELA DE RESULTADOS DO EXPERIMENTO 3

teste	Tempo de execução	Erro gerado	Erros gerados
1	3 dias 3 horas	StackOverflow	2
2	15 dias 15 horas	StackOverflow	10
3	21 dias 20 horas	StackOverflow	14
4	46 dias 21 horas	Falha de seguimentação	30
5	17 dias 4 horas	StackOverflow	11
6	14 dias 2 horas	StackOverflow	9
7	10 dias 23 horas	Preso num loop	7
8	20 dias 6 horas	defaultinterrupt()	13
9	15 dias 15 horas	defaultinterrup()	10
10	42 dias 4 horas	StackOverflow	27

A.3 TABELA DE RESULTADOS DO EXPERIMENTO 4

teste	Tempo de execução	Erro	Erros gerados
1	34 dias 8 horas	defaultinterrupt()	22
2	89 dias 2 horas	Falha de segmentação	57
3	40 dias 15 horas	Falha de segmentação	26
4	17 dias 4 horas	Falha de segmentação	11
5	26 dias 13 horas	defaultInterrupt()	17
6	31 dias 6 horas	defaultinterrupt()	20
7	60 dias 23 horas	defaultinterrupt()	39
8	40 dias 15 horas	defaultinterrupt()	26
9	93 dias 18 horas	defaultinterrupt()	60
10	28 dias 3 horas	defaultinterrupt()	18

A.4 TABELA DE RESULTADOS DO EXPERIMENTO 5

teste	Tempo de execução	Erro	Erros gerados
1	6552 dias 10 horas	StackOverflow	42
2	3432 dias 5 horas	defaultInterrupt()	22
3	936 dias 1 hora	StackOverflow	6
4	3264 dias 17 horas	StackOverflow	21
5	624 dias 2 hora	StackOverflow	4
6	6864 dias 11 horas	StackOverflow	44
7	9216 dias 3 horas	defaultinterrupt()	59
8	624 dias	No source code	4
9	6384 dias 24 horas	StackOverflow	41
10	1248 dias 2 horas	defaultinterrupt()	8