UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA E ELETRÔNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA

Ian Garner Downie

Estratégia de comunicação para configuração do escalonamento de atividades em nanossatélites

Florianópolis
2021

Ian Garner Downie

Estratégia de comunicação para configuração do escalonamento de atividades em nanossatélites

Trabalho Conclusão do Curso de Graduação em Engenharia Eletrônica do Departamento de Engenharia Elétrica e Eletrônica da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Bacharel em Engenharia Eletrônica
Orientador: Prof. Eduardo Augusto Bezerra, Dr.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Downie, Ian Garner
Estratégia de comunicação para configuração do escalonamento de atividades em nanossatélites / Ian Garner Downie ; orientador, Eduardo Augusto Bezerra, 2021.
40 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia Eletrônica, Florianópolis, 2021.

Inclui referências.

1. Engenharia Eletrônica. 2. Nanossatélite. 3. Escalonador ciente de energia . 4. Telemetria e telecomandos . I. Bezerra, Eduardo Augusto. II. Universidade Federal de Santa Catarina. Graduação em Engenharia Eletrônica. III. Título.

Ian Garner Downie

Estratégia de comunicação para configuração do escalonamento de atividades em nanossatélites

Este Trabalho Conclusão de Curso foi julgado adequado para obtenção do Título de "Bacharel em Engenharia Eletrônica" e aprovado em sua forma final pelo Curso de Graduação em Engenharia Eletrônica

Florianópolis, 29 de setembro de 2021.

Documento assinado digitalmente
Fernando Rangel de Sousa
Data: 05/10/2021 06:05:35-0300
CPF: 884.649.114-91
Verifique as assinaturas em https://v.ufsc.br

Prof. Fernando Rangel de Sousa, Dr.
Coordenador do Curso

Banca Examinadora:

Documento assinado digitalmente
Eduardo Augusto Bezerra
Data: 30/09/2021 12:13:29-0300
CPF: 830.851.577-00
Verifique as assinaturas em https://v.ufsc.br

Prof. Eduardo Augusto Bezerra, Dr.
Orientador
Universidade Federal de Santa Catarina, UFSC

Documento assinado digitalmente
Anderson Wedderhoff Spengler
Data: 30/09/2021 14:13:09-0300
CPF: 052.275.659-02
Verifique as assinaturas em https://v.ufsc.br

Prof. Anderson Wedderhoff Spengler, Dr.
Avaliador
Universidade Federal de Santa Catarina, UFSC

Documento assinado digitalmente
Felipe Viel
Data: 30/09/2021 12:36:34-0300
CPF: 082.512.689-40
Verifique as assinaturas em https://v.ufsc.br

Prof. Felipe Viel
Avaliador
Universidade do Vale do Itajai, Univali

Este trabalho é dedicado à minha esposa.

AGRADECIMENTOS

Além de agradecer ao Brasil pela recepção que tive neste país, quero agradecer à UFSC e às pessoas do curso de Engenharia Eletrônica.

Quanto a este trabalho, agradeço ao Professor Eduardo Augusto Bezerra, ao Leonardo Slongo, ao Gabriel Marcelino pelos conselhos e à toda a gente do Spacelab-UFSC por terem criado um projeto com tantos aspectos motivadores.

RESUMO

O objetivo desse trabalho é o projeto e desenvolvimento de software para o envio de listas de atividades a serem executadas pelo nanossatélite FloripaSat-2. As listas de atividades são definidas em solo, e enviadas para o satélite a partir da estação terrestre do Spacelab-UFSC. As listas enviadas possibilitam reconfigurar as atividades do satélite que serão gerenciadas por um escalonador ciente de energia. Assim, o trabalho visa prolongar a vida útil do satélite por ajudar o escalonador embarcado a usar energia mais eficientemente e ter adaptabilidade nas atividades que o satélite realiza. O processo de envio emprega uma estrutura para organizar os dados e dá ênfase na flexibilidade do programa para aceitar um arquivo de configurações independentemente do número de atividades ali contido. O software da estação terrestre inclui um programa com interface gráfica, desenvolvido na linguagem Python, para transformar bits em pulsos físicos, que acomoda o resultado desse trabalho e fornece o protocolo para organizar os dados num formato apropriado à transmissão. Ao chegar ao satélite, as rotinas do microcontrolador identificarão esses pacotes de informação e os devolverão à estrutura na qual foram inseridos na estação terrestre para que possam ser usadas para ajustar as atividades do satélite. Uma ênfase também é dada na modularidade do código a fim de facilitar a sua reusabilidade e adaptação. Finalmente, há uma discussão sobre a utilização de *bitfields* para minimizar a quantidade de bytes enviada para o satélite.

Palavras-chave: Nanossatélite 1. Escalonador ciente de energia 2. Telemetria e telecomandos 3.

ABSTRACT

The objective of this work is the design and development of software for sending lists of activities to be performed onboard the nanosatellite FloripaSat-2. The activity lists are built on the ground, and sent to the satellite from the Spacelab-UFSC ground station. This strategy allows for the reconfiguration of the activities managed by an energy-aware scheduler. Consequently, the fulfillment of this objective will prolong the satellite's lifespan as a result of consuming energy more efficiently and having more adaptability in its activities. The dispatch process employs a structure to organise the data and permits flexibility to accept files with any number of new configurations. The ground-station software includes a program with a graphical interface, developed in the language Python, that transforms bits into pulses. On arriving in the satellite, microcontroller software routines identify these information packets and return the data to the original structure so that they can be used to reconfigure the activities of the satellite. The author puts stress on developing code that is modular in order for it to be reusable and easily adapted. Finally, there is a discussion on the use of bitfields to minimise the total number of bytes that are sent to the satellite.

Keywords: Nanosatellite 1. Energy-aware scheduler 2. Telemetria e telecommands 3.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

LEO Low Earth Orbit ou Órbita Terrestre Baixa

EPS Electric Power System ou Sistema Elétrico de Potência

GUI Graphical User Interface ou Interface Gráfica do Utilizador

.so Shared Object

GTK Gimp Toolkit

XML Extensible Markup Language

CRC Cyclic Redundancy Check ou Verificação Cíclica de Redundância

SDR Radio Defined by Software ou Software Definido por Rádio

WAV Waveform Audio File Format

.csv Comma-Separated Values ou Valores Separados por Vírgula

NRZ Non-Return-To-Zero

TT&C Telemetry, Tracking & Command ou Telemetria, Rastreio e Commando

OBDH On-Board Data Handling

GPIO General Purpose Input Output

RAM Random Access Memory

SNR Signal-to-Noise Ratio ou Razão Sinal-Ruído

FIFO First In First Out

SUMÁRIO

1 INTRODUÇÃO

A classificação de satélites de acordo com o peso considera nanossatélites como sendo objetos de 1 a 10 kg. Apesar deste peso menor— os satélites considerados grandes têm aproximadamente 1000 kg — os nanossatélites são capazes de levar os subsistemas necessários para uma missão espacial para órbitas terrestres baixas (LEO — do inglês Low Earth Orbit). Um dos subsistemas é o Sistema Elétrico de Potência (EPS — do inglês Electric Power System) que tem três funções principais: a captação, o armazenamento e a distribuição de energia (SPACELAB-UFSC, 2021).

Em um nanossatélite, a área disponível para captar energia através dos seus painéis de energia solar é limitada. Ao mesmo tempo, de forma a justificar o investimento num projeto de satélite, existe a exigência de realizar a maior quantidade de atividades possível na sua vida útil. Assim, a eficiência do EPS é um fator crítico num projeto desta natureza. Uma proposta para reconciliar estas exigências é implementar um sistema que permita o escalonamento de atividades do nanossatélite a fim de maximizar a captação de energia e estender a vida útil do satélite.

Pode-se chegar a este escalonamento por meio de algoritmos de otimização que levam em conta as atividades a serem executadas pelo satélite, a energia disponível nas baterias, a periodicidade — que resulta dos parâmetros de órbita do nanossatélite — das atividades e da energia solar disponível por meio dos seus painéis solares, suas prioridades de execução e o número de vezes que o nanossatélite interage com a estação terrestre.

O escalonador de atividades ciente de energia do satélite poderá operar sem uma comunicação direta com a estação terrestre; simplesmente, podemos definir na Terra todos os parâmetros das tarefas que serão gerenciados pelo escalonador e deixar o algoritmo de escalonamento operar. No entanto, a comunicação entre o escalonador e a Terra permite mais flexibilidade na missão. Com comunicação, podemos, por exemplo, baixar a prioridade de um *payload* em relação a outro ao perceber que os dados coletados por este primeiro *payload* são

suficientes. Consequentemente, a qualidade de serviço[1] do satélite aumentará e esta flexibilidade pode até estender a sua vida útil.

Em resumo, descrevemos um sistema de escalonamento de atividades de um nanossatélite que é reconfigurável por telecomando e visa otimizar a captação de energia e a qualidade de serviço na execução das atividades; o objetivo deste trabalho é produzir um subsistema de envio e recepção de configurações de atividades de um nanossatélite.

1.1 OBJETIVOS

Nas seções abaixo estão descritos o objetivo geral e os objetivos específicos deste TCC.

1.1.1 Objetivo Geral

Como será essencial ter rotinas em software para o envio dos planos de configurações de atividades pela estação terrestre e rotinas no sistema embarcado do satélite para a recepção e o gerenciamento da execução das atividades, podemos definir os objetivos gerais para este trabalho de conclusão de curso como sendo:

o desenvolvimento de rotinas de telecomando para ser utilizado como um módulo do software para a estação terrestre para envio do plano de configurações ao nanossatélite;

o desenvolvimento de uma sub-rotina de recepção do plano de configurações para utilização no software embarcado do nanossatélite.

1.1.2 Objetivos Específicos

A partir destes objetivos gerais, é possível enunciar os seguintes objetivos específicos:

[1] No contexto de sistemas operacionais, a qualidade de serviço pode ser considerada como a alocação dos recursos necessários para satisfazer os requerimentos de várias tarefas (Juvva, 1998)

elaboração de uma estrutura vetorial (pacote) para transmitir as novas configurações a serem executadas no satélite;

definição de uma estratégia de recepção do vetor;

simulação do envio e decodificação dos planos de tarefas.

O primeiro objetivo específico tem de levar em conta o protocolo de comunicação que permite um vetor de 1 a 220 bytes. Será também necessário identificar os bits de *overhead* do vetor, que não levam informação para a execução de atividades, mas apenas para processar corretamente a informação ali contida.

O segundo objetivo específico está vinculado ao primeiro. Uma vez definida a estrutura, é possível definir as combinações de atividades em termos de número e configurações de cada atividade para que possamos formular uma estratégia de codificação que seja capaz de capturar toda a informação necessária.

2 MATERIAIS E MÉTODOS

2.1 LINUX

Os materiais necessários para a realização deste trabalho resumem-se a um microcomputador com uma instalação de uma distribuição do sistema operacional Linux que é um pacote dos componentes necessários para ter um sistema de Linux num estado operacional e cuja vantagem principal é a quantidade de software disponível gratuitamente pela licença do projeto GNU (BRYANT *et al.*, 2016). Por causa da facilidade com a qual é instalado e manipulado, a distribuição utilizada neste trabalho é o Ubuntu 20.4, instalado com a máquina virtual VirtualBox.

Todos os diretórios de Linux, que armazenam toda a informação do utilizador, partem do diretório raiz, cuja representação é /. A partir da raiz, as definições do sistema e do utilizador são guardados (NEGUS, 2020). Um exemplo disso é o local /usr/bin/, que é primariamente o local de armazenamento de executáveis no sistema. Um executável é o programa em linguagem de máquina que é executado diretamente pelo computador.

Uma distribuição do Linux é necessária por causa da interface gráfica do utilizador (GUI — do inglês *Graphical User Interface*) do terminal de comunicação do projeto FloripaSat, o Spacelab Decoder.

2.2 C

A escolha da linguagem C é induzida pela sua portabilidade, desempenho e gestão de memória. A portabilidade do C advém da sua capacidade de ser compilada para diversos microcontroladores com poucas modificações ao código; o desempenho, da compilação diretamente para um executável binário; e a gestão de memória, da possibilidade de alocar memória dinamicamente e estaticamente.

Um requisito do trabalho é ter modularidade, de forma a poder alterar ou trocar aspectos do projeto sem ter a necessidade de considerar se estas mudanças têm impactos em outros trechos do programa. Apesar da C não ser a melhor opção quando se procura modularidade — C++ pode ser uma melhor escolha — uma abordagem da separação da implementação da interface, com a *source* sendo a implementação e a *header*, a interface, faz com que tenhamos uma forma de fornecer funções para um programa-cliente por meio das interfaces (HAYES, 2001).

2.3 MAKE

Embora seja possível criar módulos em C, se o cliente não é uma outra rotina escrita em C, mas numa linguagem como Python, a forma mais fácil de acessar estes módulos é como biblioteca compartilhada, um arquivo .so (do inglês — *shared object*), que é integrado no programa no ligador para produzir o executável. Este tipo de arquivo é de uma biblioteca dinâmica, que tem a vantagem de só incluir as rotinas que são invocadas no programa e, assim, não resultam no carregamento de rotinas não chamadas no programa.

Uma forma conveniente de produzir essas bibliotecas é com a utilidade *make* e um *makefile*. Os *makefiles* são compostos de descrições das relações entre arquivos de um programa e de comandos para a utilidade *make* que, por sua vez, determina quais seções de

programas grandes precisam de recomplicação e quais não, de forma a facilitar a recomplicação de tais programas (BUFFENBERGER *et al*, 2001).

2.4 PYTHON

A Python é uma linguagem de alto nível com bibliotecas extensas que facilitam a produção de programas (PYTHON, s.d.). Devido à facilidade com a qual se produzem programas, interfaces gráficas e gráficos, a Python foi escolhida para a elaboração do Spacelab Decoder. No entanto, esta escolha implica a utilização da sua biblioteca "ctypes" como interface para poder integrar as rotinas escritas em C: o envio de novas configurações de atividades e o programa que codifica esse envio, o New Generation Ham (NGHam).

Na Python, podemos produzir funções e classes em módulos, que podem ser empacotadas em *packages*. Da mesma forma que módulos fazem com que o programador não tenha de se preocupar com os nomes de variáveis de outros módulos, *packages* distinguem dois módulos de nome igual pelo *package* de que são chamados (PYTHON, s.d.).

2.5 GTK E GLADE

A lógica da funcionalidade do Spacelab Decoder é definida na Python e rotinas específicas são feitas na C, mas a GUI é mais facilmente definida pelas ferramentas GTK e Glade.

A Glade é uma ferramenta de produção de GUIs que permite que o utilizador trabalhe diretamente na interface, sem precisar escrever o código (CONDREN *et al*. 2006). A biblioteca da interface, que interpreta e monta a interface de acordo com o que foi desenhado no Glade, é o PyGTK, que é a versão em Python do GTK (Gimp ToolKit). Por sua vez, GTK é um conjunto de ferramentas para a produção de GUIs. A Glade gera um arquivo de XML (Extensible Markup Language), um arquivo cujo formato é simultaneamente legível para máquinas e pessoas, que é interpretado e montado dentro do programa do Spacelab Decoder escrita na Python para produzir a sua GUI.

2.6 NGHAM

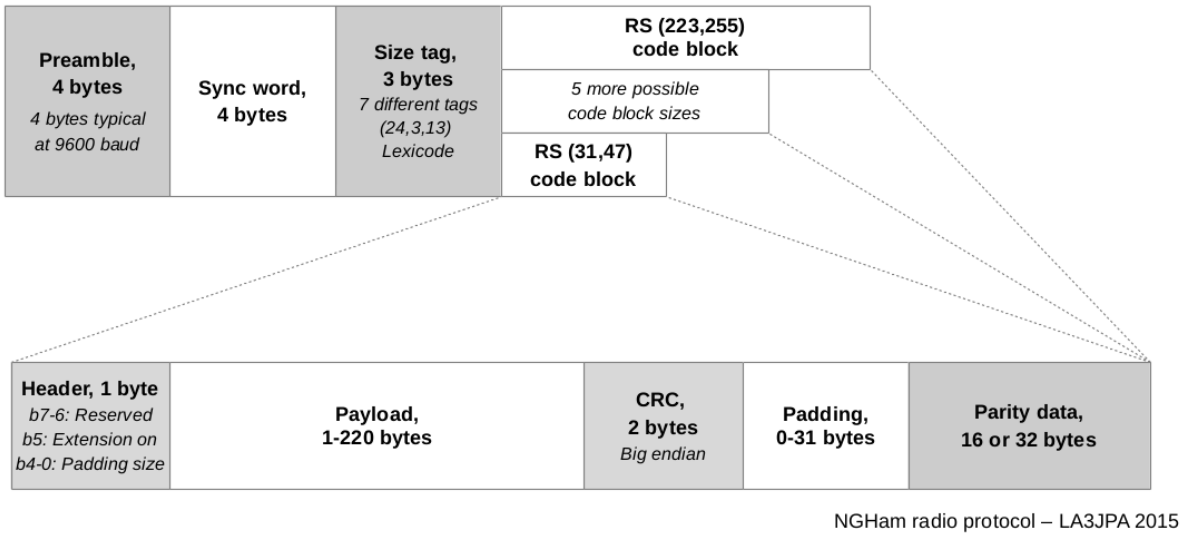
A outra biblioteca de C que é chamada pelo Spacelab Decoder é a NGHam. Apesar da NGHam já ser integrada no Spacelab Decoder, é necessário que entendamos algumas das suas caraterísticas para poder passar os dados das atividades às suas rotinas de codificação. NGHam é um protocolo da camada de enlace de dados do modelo OSI, a última camada antes da camada física, onde quadros são formados para a transmissão.

O quadro de NGHam deixa de 1 a 220 bytes para o *payload* (SKAGMO, 2014). A decisão da equipe do projeto é, dentro do *payload*, ter 1 byte para identificação, 7 bytes para a identificação do endereço fonte e 212 bytes para o conteúdo do pacote. Os 212 bytes conterão o vetor da atualização das atividades.

O NGHam é parecido com o AX.25, um protocolo também utilizado no rádio amador, no sentido que tem um formato de quadro, e utiliza a verificação cíclica de redundância (do inglês, CRC - Cyclic Redundancy Check) para evitar erros na transmissão; CRC é o resultado de um polinômio incluído em cada quadro que garante a veracidade do payload se ela é igual a zero. O que diferencia o NGHam do AX.25 é a utilização do código Reed-Solomon na codificação do *Payload*.

O formato do protocolo é mostrado na figura 1. O preâmbulo serve caso alguns bytes iniciais sejam perdidos; assim, o receptor “acorda” ao receber o preâmbulo e realiza a configuração do ganho e de outros parâmetros. Agora que o receptor está apto para aceitar dados, o *sync word* é um código conhecido que treina o receptor para sincronização de bits e de pacotes. A sincronização de bits acontece na camada física onde cada bit tem de ser distinguido do outro para conseguir comunicação e a chave para isto é a sincronização entre os relógios do emissor e o receptor. Para a sincronização de pacotes, o *sync word* tem de assinalar o início do pacote ao nível dos dados. O *size tag* fornece o tamanho do *payload*. Cada definição do tamanho é diferenciada de qualquer outra definição por 13 bits; em outras palavras, a distância de Hamming é 13 (PATTERSON *et al.*, 2014). Esta forma de codificar o *size tag* deixa-o mais robusto a erros de recepção no satélite.

Figura 1 — Protocolo de NGHam rádio



Fonte: (SKAGMO, 2014)

Dentro do *code block,* há um *header*, o CRC, *padding* e os dados de paridade de Reed-Solomon. O código Reed-Solomon serve para detectar e corrigir erros no *payload* através da inserção de bits adicionais. Desse modo, o CRC é protegido contra erros e fornece uma verificação adicional da exatidão dos dados (Levy *et al.*, 2019). Há três possíveis resultados de usar o código Reed-Solomon (CCSDS, 2020).:

1. Se $2s + r < 2t$ (s erros, r apagamentos e t é o tamanho dos dados de paridade) todos os dados vão ser recuperados e *payload* vai ser transmitido sem erro;
2. O decodificador vai detectar que não consegue recuperar todos os dados e assinalar isto;
3. O decodificador vai codificar mal os dados e não vai assinalar isto.

A probabilidade de cada um dos três resultados acontecer depende do tipo do código Reed-Solomon, definido por RS(*n,k*), sendo *n* o tamanho do bloco e *k* o tamanho para os dados, e o número e distribuição dos erros. A probabilidade do terceiro caso, com certeza o mais grave, para o RS(255, 223) usado no FloripaSat-2, é menor do que $4{,}78 \times 10^{-14}$ (CCSDS, 2020).

2.7 RÁDIO DEFINIDO POR SOFTWARE (SDR — DO INGLÊS *SOFTWARE DEFINED RADIO*)

O SDR define a utilização do software para qualquer etapa da transmissão e/ou recepção de informação que tradicionalmente seria realizada em hardware (SHARP *et al,* 1998). Essa substituição de hardware por software resulta em sistemas que são mais facilmente reprogramados, reutilizados e baratos.

Embora o SDR não seja diretamente usado no presente trabalho, é importante entender o seu papel para a integração da nova funcionalidade no Spacelab Decoder para saber como os dados devem ser entregues. O SDR da estação terrestre é o B210 da Ettus. Nesse equipamento, é possível implementar os blocos de transmissão e/ou recepção conforme as necessidades do projeto e as limitações do hardware através de criação de fluxogramas configurados do software GNUradio (MCCASKEY *et al*, 2018).

2.8 FREERTOS

Em todos os sub-sistemas do satélite, um microcontrolador MSP430 roda as aplicações do *firmware* (SPACELAB-UFSC, 2021) no sistema operativo de tempo real, o FreeRTOS. Como cada microcontrolador tem só um *core*, mas várias tarefas com deadlines fixos que têm de ser cumpridos para assegurar a operação das funções essenciais do satélite, precisamos de uma forma de organizar os recursos computacionais e físicos disponíveis. Um sistema operativo de tempo real gerencia estes recursos de acordo com a prioridade alocada para cada tarefa para que estes deadlines sejam cumpridos de forma determinística — ou seja, pré-determinada e previsível (BARRY, 2016).

2.8.1 Definição de “atividade”

É importante salientar a diferença entre o escalonador de atividades — muitas vezes chamado de escalonador de tarefas — e o escalonador de tarefas do sistema operacional. Enquanto o escalonador do sistema operacional escalona todas as tarefas que compõem o programa em execução (as *tasks*), o escalonador de atividades escalona apenas um subconjunto de tarefas de acordo com um critério que não faz parte do escopo deste trabalho.

Assim, apesar de ser chamado de escalonador em várias fontes (SLONGO *et al.,* 2018. RIGO *et al.,* 2021), de forma a distinguir um escalonador do outro, uma possível fonte de mal-entendidos numa equipe multi-disciplinar como a do FloripaSat-2, foi necessário escolher um nome para o escalonador de atividades. Assim, nomear o escalonador de atividades como Activity Manager evita que haja confusão entre pessoas formadas na Ciência de Computação, que estão cientes do escalonamento de tarefas em sistemas operacionais, e pessoas de missões espaciais, que falam de escalonamento sem perceber que o termo pode ser confundido com o escalonamento do sistema operacional. A seguinte tabela resume as diferenças no contexto do FloripaSat-2.

Tabela 1 – Resumo das diferenças entre o Escalonador e o Activity Manager

	Escalonador	**Activity Manager**
Descrição	Escalonador de tarefas (tasks/threads)	Escalonador de atividades
Posição em hierarquia	Do sistema operacional (FreeRTOS)	Tarefa escalonada pelo escalonador do sistema operacional

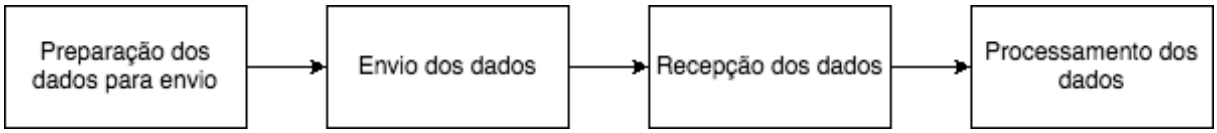
O motivo principal pelo qual o escalonador do sistema operacional não é substituído por um escalonador ciente de energia é a dificuldade que resultaria em alterar o sistema operacional utilizado pelas missões de FloripaSat, o FreeRTOS. Se o FreeRTOS fosse alterado, a sua confiabilidade, adquirida durante testes e todos os casos de uso até hoje, seria perdida (PÁEZ *et al,* 2015); teríamos de realizar novos testes e perderíamos toda a herança de voo, comprovante da confiabilidade de tecnologia em voos anteriores (ATEM DE CARVALHO, 2020). E, se realmente realizássemos esta substituição, haveria uma alteração ainda mais fundamental no FreeRTOS — a qualidade de ser preemptivo. Essa qualidade faz com que tarefas com uma prioridade mais alta do que outras são executadas primeiro (BARRY, 2016). O escalonador ciente de energia com esse papel faria com que a política de escalonamento não fosse baseada apenas nas prioridades mas também na energia disponível.

Portanto, já tendo o algoritmo de escalonamento, uma divisão de tarefas terá de ser realizada para definir quais serão escalonadas pelo Activity Manager e quais pelo escalonador do FreeRTOS. Esta definição é fora do escopo desse trabalho.

3 DESENVOLVIMENTO DO CÓDIGO

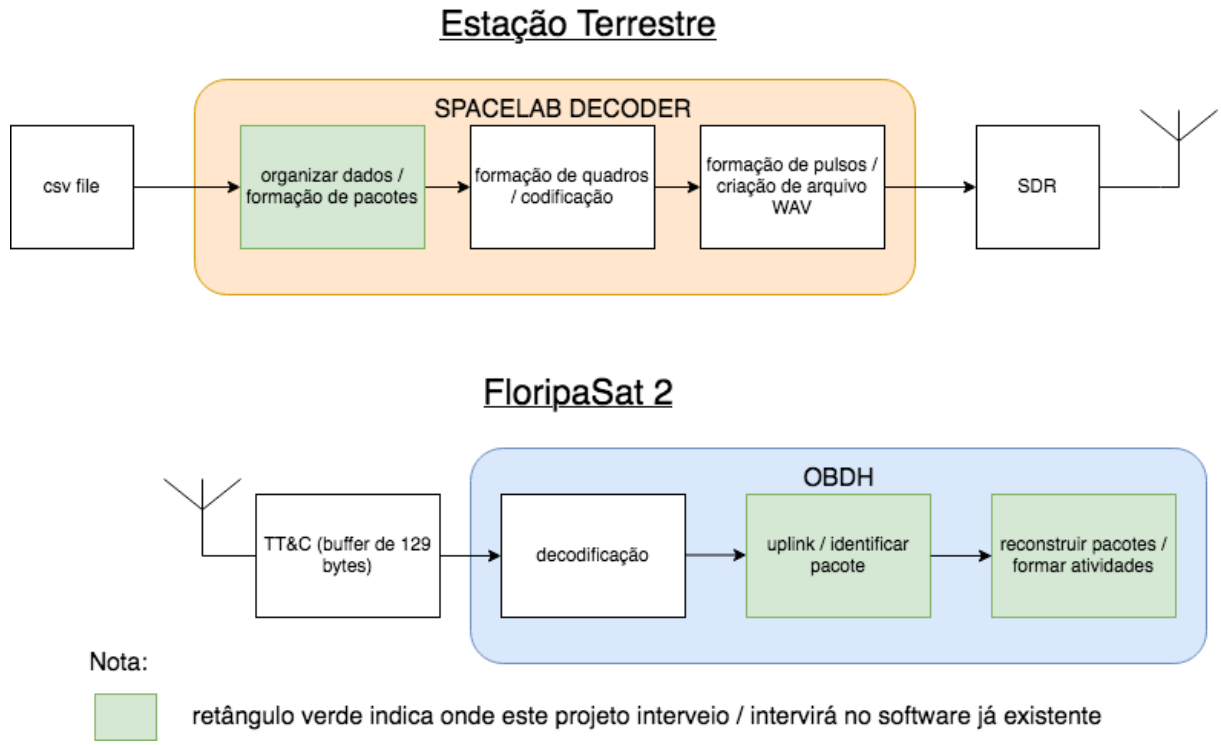
A implementação de um sub-sistema para o envio e recepção do código exige a utilização dos recursos tecnológicos descritos na seção 2 para integrá-lo nos sistemas já existentes do projeto FloripaSat-2. Vamos examinar como esses recursos são utilizados ao abordar as quatro etapas necessários para o projeto que são resumidas na Figura 2. A figura 3 resume onde esse projeto encaixa na estrutura já existente da estação terrestre e do FloripaSat-2.

Figura 2 — As quatro etapas do projeto



Fonte: Figura realizada pelo autor

Figura 3 — Resumo da intervenção



Fonte: Figura realizada pelo autor

3.1 PREPARAÇÃO DOS DADOS PARA ENVIO

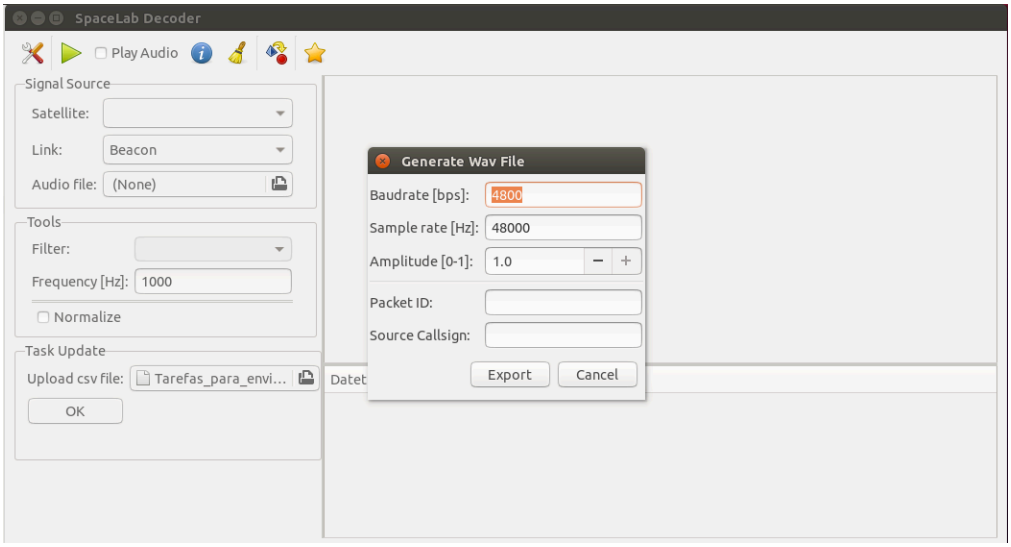
3.1.1 O Spacelab Decoder

O envio de novas configurações das atividades do FloripaSat-2 será realizado na Estação Terrestre do FloripaSat-2 a partir do Spacelab Decoder, aplicação para a codificação e a decodificação de gravações de áudio de comunicação com o satélite, que foi elaborada na linguagem Python. Como a integração do NGHam já tinha sido realizada, restou para este trabalho integrar o envio das configurações de atividades.

3.1.1.1 Interface gráfica do utilizador

A implementação da funcionalidade de atualizar as atividades do satélite exige que o Spacelab Decoder tenha acesso aos dados que o utilizador quer enviar. Consequentemente, o GUI foi atualizado para ter a opção de anexar o arquivo dos dados, *Task Update* na figura 3, e uma janela na qual são definidas a baud rate, taxa de amostragem, amplitude, *Packet ID* e *Source Callsign*; todas são usadas na produção do arquivo WAV (Waveform Audio File Format) que é compatível com o SDR, menos o *Packet ID e Source Callsign*, que são acrescentados ao pacote de dados a ser transmitido.

Figura 4 — A GUI do Spacelab Decoder com a funcionalidade da atualização de atividades



Fonte: Imagem realizada pelo autor

3.1.2 Rotinas para a preparação dos dados

O primeiro passo para a introdução desta nova funcionalidade no Spacelab Decoder é a elaboração de algumas rotinas na linguagem C que organizam os dados. A figura 4 sintetiza as funções que as rotinas desempenham para que os dados sejam preparados para o envio.

Figura 5 — Preparação dos dados para envio

Fonte: Figura realizada pelo autor

3.1.2.1 calculatePackets

Os dados são organizados numa planilha, que é convertida num arquivo .csv (Valores Separados por Vírgula — do inglês *comma-separated values*) para que a sua manipulação na C seja facilitada. De modo a padronizar o formato das configurações para cada atividade, uma *struct* é utilizada com *bit fields* com a finalidade de reduzir o tamanho dos pacotes que são enviados para o satélite. O motivo por essa redução é o fato que o FloripaSat-2 será comandado de uma única estação terrestre, com o número de acessos diários limitado a 3 ou 4, de duração de aproximadamente 9 a 14 minutos e ângulos de elevação variáveis. Essas condições sugerem que a maximização da quantidade de informação transmitida num acesso possa ser benéfica para a missão. A *struct* tem o formato mostrado na Figura 5.

Figura 6 — Formato da *struct* usada para armazenar as configurações de atividades

```
typedef struct Task {

    // BLOCK 1: 24 + 4 + 4 = 32 bits
  unsigned  int current_sig: 24; // max value is 9830399 => 2^24 = 16777216
  unsigned  int priority: 4 ; // 15 possible priorities => 2^4 = 16
  unsigned  int resource_1: 4 ; // 11 possible resources => 2^4 = 16


    // BLOCK 2: 19 + 6 + 4 = 29 bits
  unsigned  int period: 19 ; // maximum value for period is 432000 (43200000/100) => 2^19 = 524288
  unsigned  int id_code: 6; // 35 possible tasks => 2^6 = 64
  unsigned  int resource_2: 4 ; // 11 possible resources => 2^4 = 16

    // BLOCK 3: 17 + 4 + 6 = 27 bits
  unsigned int max_CPU: 17 ; // maximum max_CPU value is 120000 => 2^17 = 131072
  unsigned int current_exp: 4; // max value is 11 => 2^4 = 16
  unsigned int voltage: 6; // multiplied by 10 so max = 50 => 2^6 = 64

} Task;
```

Fonte: Imagem realizada pelo autor

Uma vez que os *bit fields* permitem que apenas os bits necessários para cada variável sejam alocados e tendo em conta que a definição de uma variável como, por exemplo, um *unsigned int* resulta na criação de um espaço na memória de 4 bytes, podemos reduzir o número de bytes necessário ao declarar as variáveis numa sequência que não resulta num total de bits maior que 32. Por este motivo, “blocos” foram criados para que cada nova configuração de uma atividade só ocupe 12 bytes, ou 3 *unsigned ints*. As variáveis *current* e *voltage* não representam números inteiros e precisam de processamento adicional. No caso de *current*, usamos a representação de vírgula flutuante com base de 2. Como os valores representados por *voltage* são não-inteiros entre 0 e 10, optamos por multiplicar o valor por 10 de modo que seja possível armazenar os valores como inteiros. Nos dois casos, os valores originais são reconstruídos na recepção no satélite.

Tendo em conta que o número de novas configurações de atividades pode variar de um envio para outro, a decisão foi tomada de contar o número de linhas no arquivo .csv de modo a deixar esta funcionalidade mais flexível. No entanto, esta flexibilidade não é viável para a memória do microcontrolador no satélite, que poderia ter problemas graves de *overflow* se arrays não tivessem um tamanho fixo alocado estaticamente. Assim, de forma a contornar esta inconsistência, um limite superior para o número de novas configurações terá de ser estabelecido.

Sabendo o tamanho de uma atividade, o número de atividades e o tamanho do *payload* permitido num pacote NGHam, podemos definir o número de atividades em cada pacote. Assim, um pouco de artimética simples divide as atividades entre pacotes. A linha 6 da figura 6 retorna o número de pacotes e como o resultado de linha 7 é um *int,* o seu valor é um inteiro mesmo que o resultado artimético seja não-inteiro. Assim, conseguimos dividir as atividades da forma mais igual possível entre os pacotes, ainda que admitamos a possibilidade de ter pacotes com uma atividade a mais do que outros, por causa de um eventual número total de atividades não divisível pelo número total de pacotes definido na linha 6. A linha 9 assegura que sejam conhecidos quantos pacotes vão ter uma atividade a mais.

Figura 7 — Cálculo do número e do tamanho dos pacotes

```
1  float numberRowsf = countRows(fp1);
2  int taskSize = 96; // in bits (12 bytes)
3  float totalSize = numberRowsf*taskSize/8; // in bytes
4  int maxPacketSize = 204; // (212 - 8) / 12 = 17; 204 is the
5  //largest number divisible by 12 and less than 212
6  *numberPackets = ceil(totalSize/maxPacketSize);
7  *minPacketSize = numberRowsf/(*numberPackets);
8  *numberRows = (int)numberRowsf;
9  *numberBonusPackets = *numberRows%*minPacketSize;
10 // packets that will include an extra task
```

Fonte: Imagem realizada pelo autor

3.1.2.2 organiseTasks

Essa rotina tem o papel de tirar os dados do arquivo .csv e depositá-los num formato que facilita o seu envio, de acordo com os dados fornecidos pela rotina *calculatePackets*. Como o NGHam processa variáveis de um byte, *uint8_t*, e não de 4 bytes, como *unsigned int*, temos de deixá-los neste formato. Assim, usamos máscaras de bits e deslocamentos lógicos para organizar os dados no formato que o NGHam exige, como mostra o trecho da rotina na figura 6.

Figura 8 — A utilização de máscaras de bits e deslocamentos lógicos para organizar dados

```
packetBuf[j][3*i+1] = (task[k+i].resource_1 & 0x0000000F);
packetBuf[j][3*i+1] |= (task[k+i].current_sig << 8);
packetBuf[j][3*i+1] |= (task[k+i].priority << 4);

packet[j][12*i+6] = packetBuf[j][3*i+1];
packet[j][12*i+7] = packetBuf[j][3*i+1] >> 8;
packet[j][12*i+8] = packetBuf[j][3*i+1] >> 16;
packet[j][12*i+9] = packetBuf[j][3*i+1] >> 24;
```

Fonte: Imagem realizada pelo autor

Na elaboração deste trabalho, um "cliente" de C tomou inicialmente o lugar do Python como uma etapa intermediária no desenvolvimento para receber os "j" *packets* na figura 6, que representam novos pacotes a ser enviados ao satélite. De forma a deixar esses pacotes disponíveis para o cliente, sem ter comportamento inesperado, é necessário alocar memória no *heap* quando se desenvolve na C. No entanto, no Python, o gerenciamento de memória é realizada pelo gerenciador de Python. Mas, como o array de duas dimensões *packet* é declarado na C, temos de seguir as melhores práticas de C, que implicam a alocação e remoção de memória no *heap*, que assegura ao array estar disponível depois de uma chamada à função onde *packet* é declarado.

3.1.2.3 sendData

Essa rotina serve para transferir os dados do array declarado em C para um array declarado em Python. Visto que há outros dados que são fornecidos pelo utilizador do Spacelab Decoder, como o *Packet ID,* a quais temos de anexar o *payload* antes de chamar o NGHam, o *payload* tem de ser convertido numa lista de Python. Por este motivo, criamos um array de bytes de ctype no Python, a partir do qual podemos produzir um ponteiro, que, por sua vez, pode ser passado para a rotina *sendData*. Agora que os dados estão acessíveis no Python, podemos convertê-los numa lista de Python e anexá-los no pacote final, junto com os outros dados do Spacelab Decoder, que é entregue ao NGHam.

3.1.3 Compilação do Spacelab Decoder

Além de alterar a GUI, para integrar a nova biblioteca no Spacelab Decoder, foi necessário modificar os makefiles da aplicação. No caso da nova biblioteca para nossa funcionalidade, foi necessário criar um novo alvo que é um objeto compartilhado.

Depois da criação da biblioteca, esta é chamada na rotina de Python *spacelabdecoder.py*. Esta rotina faz parte do *package spacelab-decoder* e implementa as funcionalidades do Spacelab Decoder. O motivo pelo qual este *package* foi criado é a possibilidade de usar um *launch script*, que é um código que lança o programa. Uma forma de implementar um *launch script* é com o mecanismo de *entry points* do *setuptools* num arquivo *setup.py*. O *entry point* tem o seguinte formato:

nome do executável = módulo.com:função_para_criar

No nosso caso, temos:

spacelab-decoder = spacelab_decoder.__main__:main

Assim, procuramos no arquivo *__main__*, na pasta *spacelab_decoder*, que contém a função *main*, que chama a rotina *SpaceLabDecoder* que implementa a GUI e as suas funcionalidades, incluindo a de atualizar as atividades do satélite. O executável criado é no diretório raiz, no local */usr/bin*, onde estão guardadas aplicações fornecidas pelo sistema operacional. Este executável é chamado dentro um arquivo *.desktop*, um atalho para que a aplicação apareça no menu de aplicações de Linux.

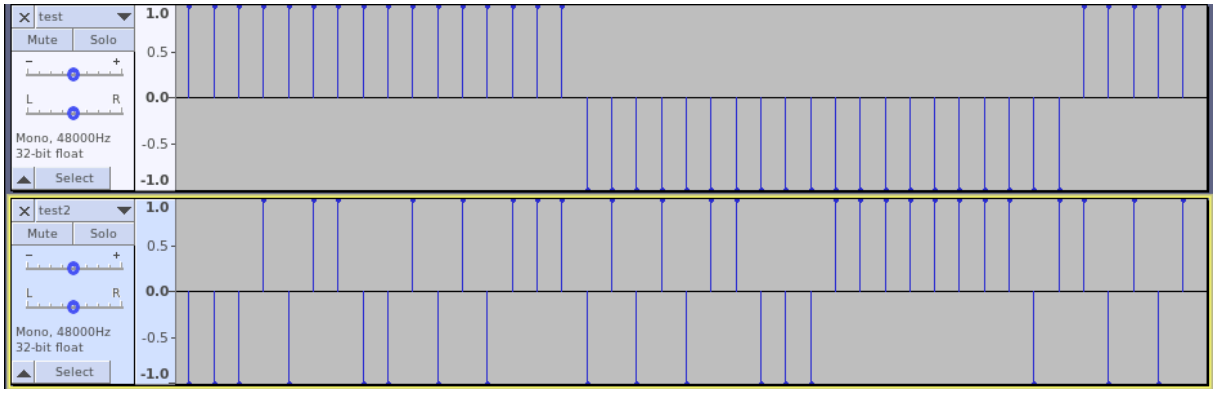
3.2 ENVIO DOS DADOS

Os quadros formados pelo NGHam são convertidos em arquivos WAV, que são streams de bits e são aceitos como a fonte para o SDR. A baud rate, taxa de amostragem, e a amplitude fornecidas pelo operador através da GUI são utilizadas na geração do arquivo. O quadros são transmitidos na banda UHF a 450 MHz.

Para realizar a transmissão de informação digital, os valores binários têm de ser transformados em pulsos físicos que podem ser transmitidos. A estação terrestre do Spacelab poderia realizar o processo de formação de pulsos dentro do SDR; porém, isto faria com que qualquer rádio amador que quisesse comunicar com o satélite precisaria de SDR com transmissão. De forma a permitir que rádio amadores possam comunicar com o satélite, o Spacelab Decoder cria um arquivo WAV com uma rotina que torna os valores binários em pulsos apenas com algumas linhas de código e os valores de *Sample rate* e Baud rate que são fornecidos pelo operador no Spacelab Decoder. Por exemplo, como a Baud rate de FloripaSat-2 é 9,6 kbps, se o operador fornecer um *Sample rate* 48 kHz, cada valor binário vai ser transformado em cinco — como se usássemos os comandos de Matlab UPSAMPLE e FILTER com um pulso retangular (KOVALEVSKA, 2008). Ainda, como a codificação de pulsos a ser usada é *non-return-to-zero* (NRZ), os 0s são transformados em -1 de amplitude.

A figura 8 mostra a diferença na formação de pulso quando taxas de amostragem de 48kHz e 9,6kHz são usadas. Podemos notar que quando a taxa de amostragem é igual a baud, como na sequência inferior, reproduzimos cada pulso apenas uma vez; a sequência superior reproduz cada pulso cinco vezes. A gravação que resulta deste processo pode ser modulada na frequência de 450 MHz e transmitida com qualquer programa de SDR.

Figura 9 — Sequências de bits com taxas de amostragem de 48kHz (superior) e de 9,6kHz (inferior)



Fonte: Imagem realizada pelo autor

3.3 RECEPÇÃO DOS DADOS

O quadro será recebido no receptor do satélite no módulo de Telemetria, Rastreio e Commando (TT&C — do inglês Telemetry, Tracking Command) e o pacote de dados redirecionado ao Subsistema de Supervisão de Bordo (OBDH — do inglês On-Board Data Handling) (SPACELAB-UFSC, 2021). No OBDH, ao identificar o *Packet ID* associado à atualização das configurações das atividades na rotina "uplink", o pacote será encaminhado para o processamento. A rotina "uplink" tem uma limitação de só poder receber 129 bytes a cada vez por causa do tamanho do buffer interno em hardware do rádio (SILICON LABS, 2014). Ao ter seu buffer preenchido, o rádio assinala num GPIO (Entrada/Saída de Propósito Geral — do inglês *General Purpose Input/Output*) que o buffer tem de ser esvaziado e os dados são transferidos. Isto procede até que todos os dados sejam transferidos.

Como foi mencionado na seção 3.1, um limite superior tem de ser estabelecido para a quantidade de configurações que pode ser enviada para o satélite por causa da capacidade de memória no microcontrolador MSP430F6659, onde há 64 kB de RAM, que será utilizada para o *heap* e o *stack.* Das cinco possíveis configurações de *heap* que o FreeRTOS disponibiliza, a equipe de FloripaSat-2 escolheu o Heap_1, no qual só é possível criar tarefas antes do começo do escalonador (BARRY, 2016). O motivo para essa escolha é não permitir a alocação dinâmica de memória, visto que isto poderia resultar em incertezas temporais e de memória que seriam críticas para a operação do satélite.

Quando o Activity Manager for implementado no FloripaSat-2, será uma tarefa que o FreeRTOS gerencia no seu escalonador. Embora ainda não tenha sido definido, é possível especular que as configurações das atividades que serão escalonadas pelo Activity Manager estarão armazenadas permanentemente na memória flash do OBDH e possivelmente na RAM do MSP430F6659 de forma temporária. Esta decisão terá de levar em conta o tempo que demora para buscar as configurações, o consumo de energia e a segurança dos dados (NASA, 2021). No entanto, como a RAM é volátil e a memória flash não é, a segurança dos dados entre *resets*, programados ou não, só será garantida se algum espaço for alocado na memória flash para esta finalidade. Quanto à rotina de receber os dados, a sua memória poderá ser

alocada no *stack*, uma vez que só vai ser preciso durante a chamada pela “uplink”; os dados serão transferidos para a flash logo depois da recepção.

Em termos do desenvolvimento do software para realizar essa função, na altura do desenvolvimento, não havia todo o software que implementa o OBDH disponível, o que limitou a elaboração completa das rotinas de recepção. No entanto, um cenário foi criado com base no envio de 35 novas configurações, que resulta em 3 pacotes a serem recebidos e processados. A rotina de recepção é modelada na preparação dos pacotes para envio; entretanto, foi possível reduzir o código e torná-lo mais modular. Também, o *buffer* usado é um terço do tamanho daquele que é usado na estação terrestre, o que diminui o tamanho do *stack* da rotina. Testes mostram que o código é capaz de receber pacotes pelo uplink e organizar de novo os dados na estrutura original, *Task*, mostrada na figura 5. A partir desse ponto, as novas configurações podem ser distribuídas para as atividades do satélite.

3.4 PROCESSAMENTO DE DADOS

O processamento dos dados resultará na reconfiguração das atividades gerenciadas pelo Activity Manager. Como a lista completa de atividades delegada ao Activity Manager será no *boot*, cada atividade no envio tem de ser identificada e as suas configurações têm de ser atualizadas.

4. RESULTADOS

É interessante examinar a escolha de usar *bitfields* no lugar de bytes como o tamanho mínimo para um elemento no *struct* que reúne as configurações para uma dada atividade. A tabela 2 resume a diferença e mostra que esta escolha resulta numa redução de um terço na quantidade de dados que tem de ser enviada para uma nova configuração.

Tabela 2 — Número de bytes necessário para organizar uma *struct* com *bitfields* e com bytes

	Como *bitfields* (bits)	**Como bytes (bits)**
current_sig	24	32
priority	4	8
resource_1	4	8
period	19	32
id_code	6	8
resource_2	4	8
max_CPU	17	32
current_exp	4	8
voltage	6	8
Total Bytes	**96**	**144**

Salamanca (2020), na sua tese de doutorado, constrói dois cenários de transmissão da estação terrestre para o FloripaSat-1, o primeiro satélite do Spacelab da UFSC: o primeiro é com retransmissão com codewords de um tamanho fixo e o segundo, com tamanho variável. Para os dois cenários, também considera duas razões sinal-ruído (SNR — do inglês *Signal-Noise Ratio*) na recepção dos quadros no satélite: 5dB e 20 dB. Para todos os casos, uma taxa de transmissão de 250 kbps é utilizada. O autor utiliza o dia 12 de julho de 2020, quando houve quatro passagens do satélite pela estação terrestre, para analisar o tempo necessário para transmitir uma determinada quantidade de dados para o satélite.

Adaptando o trabalho de Salamanca, podemos avaliar se a utilização de *bitfields* tem um impacto crítico na recepção ou não de uma transmissão completa de novas configurações, em comparação com a unidade mínima de dados sendo um byte. Das passagens realizadas no dia observado — cujos tempos de passagem são apresentados na tabela 3 e os tempos necessários para completar um transmissão para diferentes SNRs, formas de transmissão e configurações de dados são listados na tabela 4 — a redução da quantidade de bits transmitida

não teria um impacto que alterasse a viabilidade de transmissão; apenas o tempo necessário para a transmissão cairia proporcionalmente com a redução. No entanto, como a transmissão é inviável para três dos quatro cenários de SNR e forma de transmissão quando o ângulo máximo do satélite é 6º, vemos que haverá um limiar entre a viabilidade e a inviabilidade. Portanto, a redução na quantidade de dados viabilizará o envio de dados para certos sistemas de retransmissão, ângulos máximos de passagem e SNRs; mais dados seriam necessários para definir exatamente a fronteira entre a viabilidade e a inviabilidade. Em adicão, como a capacidade de processamento dentro de satélites é um recurso muitas vezes escasso, poderá ser interessante comparar se o uso de *bitfields* compensa quando se leva em conta o processamento adicional que os *bitfields* impõem em termos das máscaras de bits e dos deslocamentos lógicos necessário, como visto anteriormente.

Tabela 3 — O tempo de passagem de três dos quatro cenários observados no dia 12 de julho de 2020

Ângulo de elevação máxima	**Tempo de passagem (seg)**
6º	649
22º	800
72º	865

Tabela 4 — Tempo hipotético necessário (em segundos) para a estação terrestre entregar com sucesso as novas configurações de tarefas em formatos e sob regimes diferentes ao FloripaSat-1. Tempos de envio não viáveis são em itálico.

	5 dB				**20 dB**			
	Retransmissão fixa		**Retransmissão variável**		**Retransmissão fixa**		**Retransmissão variável**	
Ângulo max	***bitfields***	**bytes**	***bitfields***	**bytes**	***bit fields***	**bytes**	***bitfields***	**bytes**
6º	**inf**	**inf**	**$4{,}65 \times 10^{13}$**	**$6{,}20 \times 10^{13}$**	**$2{,}03 \times 10^{5}$**	**$2{,}71 \times 10^{5}$**	0,985	1,31
22º	**$6{,}68 \times 10^{15}$**	**$8{,}91 \times 10^{15}$**	9,20	12,3	0,0645	0,0861	0,0623	0,0831
72º	0,126	0,168	0,116	0,154	0,0456	0,0608	0,449	0,598

5. CONCLUSÃO

Este trabalho conseguiu desenvolver e integrar o software necessário para realizar o envio de novas configurações de atividades para o nanossatélite FloripaSat-2. Porém, o teste de envio foi limitado à abertura do arquivo .csv até a produção do arquivo WAV, que é o arquivo usado na transmissão pelo SDR. Embora as rotinas consigam processar os dados, não há verificação dos valores do arquivo .csv para confirmar que são no intervalo esperado para as atividades. Quando houver esses intervalos, eles serão facilmente incorporados nas rotinas e darão mais robustez ao sistema.

Em adição aos testes que têm de ser realizados, a recepção precisa ser integrada com o software do OBDH, sobretudo para lidar com o FIFO (primeiro a entrar, primeiro a sair — do inglês *First In, First Out*) buffer de 129 bytes. Como o escalonador de atividades ainda não foi integrado como uma tarefa do escalonador do sistema operacional, a substituição das configurações ainda resta como uma tarefa para a próxima fase da implementação do escalonador no FloripaSat-2.

Finalmente, o código foi desenvolvido de forma a adaptar-se ao número de atividades que o operador quererá enviar para o satélite. Entretanto, poderá ser mais sensato limitar essa flexibilidade se for possível definir exatamente o número de atividades que será gerenciado pelo escalonador ciente de energia. Um processo mais rígido nesse respeito seria mais simples e menos exposto a erros.

REFERÊNCIAS

JUVVA, K. Quality of Service. **Carnegie Mellon University**, 1998. Disponível em: <https://users.ece.cmu.edu/~koopman/des_s99/quality_of_service/index.html>

BRYANT, R. E.; O'HALLARON, D. R. **Computer Systems: A Programmer's Perspective, Third Edition**. [*S.l.*]. Pearson, 2016

NEGUS, C. **Linux Bible, Tenth Edition**. Indianapolis: Wiley, 2020.

HAYES, J. Modular Programming in C. **Embedded**, Nov 2001. Disponível em: <https://www.embedded.com/modular-programming-in-c/>

BUFFENBURGER, J.; GRUELL, K. A Language for Software Subsystem Composition. Ago 2002. **Proceedings of the 34th Hawaii International Conference on Science Systems**. Disponível em: < https://doi.org/10.1109/HICSS.2001.927267>

PYTHON. **Packages**. [*S.l. ; s.n.; s.d.*]. Disponível em: <https://docs.python.org/3/tutorial/modules.html#packages>

CONDREN, J.; SEUNGWON, A. Automation of Transmission Planning Analysis Process Using Python and GTK+. Out 2006. **2006 IEEE Power Engineering Society General Meeting**. Disponível em: <https://ieeexplore.ieee.org/document/1709495>

Spacelab-UFSC. **FloripaSat-2 Documentation**. Florianópolis. Set 2021. Disponível <em: https://github.com/spacelab-ufsc/floripasat2-doc>

Spacelab-UFSC. **OBDH 2.0 Documentation**. Florianópolis. Set 2021. Disponível em: <https://github.com/spacelab-ufsc/obdh2/blob/master/doc/slb-obdh2-doc-v0.7.pdf>

SKAGMO, J. P. **NGHam Manual**. [*S.l.*] Dez 2014. Disponível em: <https://github.com/skagmo/ngham/blob/master/documentation/ngham_manual.pdf>

PATTERSON, D. E. ; HENNESSY, J. L. **Computer Organization and Design: The Hardware/Software Interface, Fifth Edition**. [*S.l.*]. Morgan Kaufman, 2014.

LEVY, S; FERREIRA, K. B. Space Efficient Reed-Solomon Encoding to Detect and Correct Pointer Corruption. **Center for Computing Research, Sandia National Laboratories**, 2019. Disponível em: < https://www.osti.gov/servlets/purl/1641289>

CCSDS. **TM Synchronization and Channel Coding — Summary of Concept and Rationale**. 2020. Disponível em: <https://public.ccsds.org/Pubs/130x1g3.pdf>

BARRY, R. **Mastering the FreeRTOS™ Real Time Kernel: A Hands-On Tutorial Guide**. Real Time Engineers Ltd, 2016. Disponível em: <https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf>

SLONGO, L.K. *et al*. Energy-driven scheduling algorithm for nanosatellite energy harvesting maximization. **Acta Astronautica**, 147 (2018), p. 141 – 151. Disponível em: <https://doi.org/10.1016/j.actaastro.2018.03.052>

RIGO, C. A. et al. A nanosatellite task scheduling framework to improve mission value using fuzzy constraints. **Expert Systems With Applications**, 175 (2021). Disponível em: <https://doi.org/10.1016/j.eswa.2021.114784>

PÁEZ, F. E. FreeRTOS User Mode Scheduler for Mixed Critical Systems. Out 2015. **Sixth Argentine Conference on Embedded Systems (CASE)**. Disponível em: < 10.1109/SASE-CASE.2015.7295845>

ATEM DE CARVALHO, R. et al. **Nanosatellites: Space and Ground Technologies, Operations and Economics**. Hoboken, NJ: Wiley, 2020.

SHARP, B.A. *et al*. The Design of an Analogue RF Front End for a Multi-Role Radio. Out 1998. **IEEE Military Communications Conference.** Disponível em: < 10.1109/MILCOM.1998.722593>

MCCASKEY, M. *et al*. Machine Learning Applied to an RF Communication Channel. Dez 2018. **NAECON 2018 — IEEE National Aerospace and Electronics Conference**. Disponível em: <https://doi.org/10.1109/NAECON.2018.8556708>

KOVALEVSKA, N. **Simulation of Adjacent Channel Interference Cancellation**. Oslo. 2008. Tese de mestrado para obtenção do título de mestre em Física da Universidade de Oslo.

SILICON LABS. **Si4463/61/60-C High Performance, Low-Current Transreceiver**. [*S.l.; s.n.; s.d.*]. Disponível em: <https://www.silabs.com/documents/public/data-sheets/Si4463-61-60-C.pdf>

NASA. **Command and Data Handling**. Jan 2021. Disponível em: <https://www.nasa.gov/smallsat-institute/sst-soa-2020/command-and-data-handling>

SALAMANCA, J. J. L. **Modelo para a caraterização de canal através de Cadeias de Markov de um sistema de comunicação entre CubeSat e estação terrestre**. Florianópolis. 2020. Tese de doutorado para obtenção do título de doutor em Engenharia Elétrica da Universidade Federal de Santa Catarina.