

DANIEL BARBOSA SILVA COSTA

**Transferência de dados utilizando
comunicação por luz visível junto à plataforma
Arduino**

**São José dos Campos
15 de fevereiro de 2022**

DANIEL BARBOSA SILVA COSTA

**Transferência de dados utilizando comunicação por luz
visível junto à plataforma Arduino**

Trabalho de Conclusão de Curso apresentado ao Instituto de Ciência e Tecnologia da Universidade Federal de São Paulo - UNIFESP, como parte das atividades para obtenção do título de Bacharel em Engenharia da Computação.

Universidade Federal de São Paulo
Instituto de Ciência e Tecnologia
Bacharel em Engenharia da Computação

Orientador Prof. Dr. André Marcorin de Oliveira
Coorientadora Profa. Dra. Fernanda Quelho Rossi

São José dos Campos
15 de fevereiro de 2022

Costa, Daniel Barbosa Silva

Transferência de dados utilizando comunicação por luz visível junto à plataforma Arduino/ Daniel Barbosa Silva Costa. – São José dos Campos, 2022.

123 p.

Orientador: Prof. Dr. André Marcorin de Oliveira

Coorientadora: Profa. Dra. Fernanda Quelho Rossi

Trabalho de Conclusão de Curso – Universidade Federal De São Paulo
Instituto de Ciéncia e Tecnologia

Curso de Graduação em Engenharia da Computação, 2022.

1. Comunicação por luz visível. 2. VLC. 3. Arduino. 4. Transferência de dados I. Prof. Dr. André Marcorin de Oliveira. II. Profa. Dra. Fernanda Quelho Rossi. III. Universidade Federal de São Paulo. IV. Curso de Graduação em Engenharia da Computação. V. Transferência de dados utilizando comunicação por luz visível junto à plataforma Arduino.

DANIEL BARBOSA SILVA COSTA

Transferência de dados utilizando comunicação por luz visível junto à plataforma Arduino

Trabalho de Conclusão de Curso apresentado ao Instituto de Ciência e Tecnologia da Universidade Federal de São Paulo - UNIFESP, como parte das atividades para obtenção do título de Bacharel em Engenharia da Computação.

Trabalho aprovado. São José dos Campos, 15 de fevereiro 2022:

Prof. Dr. André Marcorin de Oliveira
Orientador

Profa. Dra. Fernanda Quelho Rossi
Coorientadora

Prof. Dr. Sérgio Ronaldo Barros dos Santos
Convidado 1

Prof. Dr. Lauro Paulo da Silva Neto
Convidado 2

São José dos Campos
15 de fevereiro de 2022

RESUMO

O projeto tem como objetivo realizar uma análise do desempenho da transmissão digital de dados, utilizando um sistema de comunicação por luz visível desenvolvido junto à plataforma Arduíno. O transmissor é composto por um LED de alto brilho, transmitindo os caracteres alfanuméricos (*byte*) em um pacote de 10 bits (1 *bit* de início, 8 *bits* de dados e 1 *bit* de parada) por chaveamento liga-desliga. O receptor é formado por um módulo de fototransistor sensível à luz visível, em que a tensão de saída é inserida em um amplificador operacional, em modo comparador, para determinar se o nível lógico do valor recebido é um valor alto ou um valor baixo. A partir disso, foram coletados os resultados gerados pelo receptor variando, em sequência, a frequência de transmissão, a distância entre o transmissor e receptor, a quantidade de carga de *bytes* transmitidas e a tensão de referência do comparador, essa última manipulada via um divisor de tensão. Ao realizar uma análise acerca dos dados coletados, as variáveis de real impacto foram com relação a tensão de referência do comparador, a distância entre transmissor e receptor e a frequência de transmissão.

Palavras-chave: Comunicação por luz visível. VLC. Arduino. Transferência de dados.

ABSTRACT

The project aims to analyze the performance of digital data transmission using a visible light communication system, developed with the Arduino platform. The transmitter consists of a high-brightness LED, transmitting alphanumeric characters (*byte*) in a 10-bit packet (1 *start bit*, 8 *data bits* and 1 *stop bit*) by on-off keying. The receiver is formed by a phototransistor module sensitive to visible light, in which the output voltage is fed into an operational amplifier, in comparator mode, to determine if the value from the logical level received is a high value or a low value. From this, the results generated by the receiver were collected, varying, in sequence, the transmission frequency, the distance between the transmitter and the receiver, the amount of *bytes* load transmitted and the reference voltage of the comparator, this last manipulated via a voltage divider. When performing an analysis of the collected data, the real impact variables were in relation to the comparator reference voltage, the distance between transmitter and receiver and the transmission frequency.

Keywords: Visible light communication. VLC. Arduino. Data transfer.

LISTA DE FIGURAS

Figura 1 – Radiofrequências e aplicações	28
Figura 2 – Comprimentos de onda e bandas de frequência da luz visível	29
Figura 3 – Esquema básico de um sistema de comunicação	29
Figura 4 – Esquemático entre comunicação serial e paralela	30
Figura 5 – Pacote de dados enviados pela comunicação por <i>UART</i>	32
Figura 6 – Arquitetura base da VLC	34
Figura 7 – Diagrama em alto nível do projeto	41
Figura 8 – Diagrama em alto nível do transmissor	42
Figura 9 – Diagrama em alto nível do transmissor	42
Figura 10 – Pacote de dados utilizado no projeto	43
Figura 11 – Máquina de estados para o módulo transmissor	44
Figura 12 – Exemplo de amostragem feita pelo receptor	45
Figura 13 – Máquina de estados para o módulo receptor	46
Figura 14 – Esquemático e implementação física do <i>hardware</i> transmissor	47
Figura 15 – Esquemático da implementação inicial do <i>hardware</i> receptor com LDR	52
Figura 16 – Esquemático da implementação inicial do <i>hardware</i> receptor com Módulo TEMT6000	53
Figura 17 – Esquemático da implementação final do <i>hardware</i> receptor com divisor de tensão feito com $R_1 = 1\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$	54
Figura 18 – Esquemático da implementação final do <i>hardware</i> receptor com divisor de tensão feito com $R_1 = 10\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$	55
Figura 19 – Esquemático da implementação final do <i>hardware</i> receptor com divisor de tensão feito com $R_1 = 50\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$	55
Figura 20 – Implementação final física do <i>hardware</i> receptor com divisor de tensão feito com $R_1 = 50\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$	56
Figura 21 – Esquemático da árvore de diretórios e arquivos gerada com os dados coletados	63
Figura 22 – Resultado gerado pelo Código F.2	68
Figura 23 – Resultado gerado pelo Código F.3	68
Figura 24 – Resultados para divisor de tensão feito com $R_1 = 1\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$ e carga de dados de 100 <i>bytes</i>	71
Figura 25 – Resultados para divisor de tensão feito com $R_1 = 1\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$ e carga de dados de 500 <i>bytes</i>	73

Figura 26 – Resultados para divisor de tensão feito com $R_1 = 1 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$ e carga de dados de 1000 <i>bytes</i>	75
Figura 27 – Resultados para divisor de tensão feito com $R_1 = 10 \text{ k}\Omega$ e $R_2 = 1 \text{k}\Omega$ e carga de dados de 100 <i>bytes</i>	77
Figura 28 – Resultados para divisor de tensão feito com $R_1 = 10 \text{ k}\Omega$ e $R_2 = 1 \text{k}\Omega$ e carga de dados de 500 <i>bytes</i>	79
Figura 29 – Resultados para divisor de tensão feito com $R_1 = 10 \text{ k}\Omega$ e $R_2 = 1 \text{k}\Omega$ e carga de dados de 1000 <i>bytes</i>	81
Figura 30 – Resultados para divisor de tensão feito com $R_1 = 50 \text{ k}\Omega$ e $R_2 = 1 \text{k}\Omega$ e carga de dados de 100 <i>bytes</i>	83
Figura 31 – Resultados para divisor de tensão feito com $R_1 = 50 \text{ k}\Omega$ e $R_2 = 1 \text{k}\Omega$ e carga de dados de 500 <i>bytes</i>	85
Figura 32 – Resultados para divisor de tensão feito com $R_1 = 50 \text{ k}\Omega$ e $R_2 = 1 \text{k}\Omega$ e carga de dados de 500 <i>bytes</i>	87

LISTA DE QUADROS

Quadro 1 – Configurações feitas via registrador EICRx	37
Quadro 2 – Configurações feitas via registrador TCCRnx	37
Quadro 3 – Configurações feitas via registradores TIMSKn	38
Quadro 4 – Configurações feitas via registradores TIFRn	38
Quadro 5 – Nomenclatura de uma transmissão com base na taxa de erro a nível de <i>bytes</i>	64
Quadro 6 – Consideração acerca do conjunto de 10 transmissões	64

LISTA DE ABREVIATURAS E SIGLAS

CPU	<i>Central Processing Unit</i>
GPIO	<i>General Purpose Input Output</i>
IoT	<i>Internet of Things</i>
ISR	<i>Interrupt Service Routine</i>
MIMO	<i>Multiple-Input-Multiple-Output</i>
UART	<i>Universal Asynchronous Receiver/Transmitter</i>
VLC	<i>Visible Light Communication</i>
WiFi	<i>Wireless Fidelity</i>

LISTA DE SÍMBOLOS

λ_p Onda de Pico de Sensibilidade

SUMÁRIO

1	INTRODUÇÃO	21
1.1	CONTEXTUALIZAÇÃO	21
1.2	TRABALHOS RELACIONADOS E APLICAÇÕES	22
1.3	OBJETIVOS	24
1.3.1	Objetivo Geral	24
1.3.2	Objetivos Específicos	24
1.4	ORGANIZAÇÃO DA MONOGRAFIA	25
2	FUNDAMENTAÇÃO TEÓRICA	27
2.1	ONDAS ELETROMAGNÉTICAS	27
2.1.1	Ondas de Rádio	27
2.1.2	Luz Visível	28
2.2	SISTEMAS DE COMUNICAÇÃO	29
2.3	COMUNICAÇÃO DIGITAL	30
2.3.1	Frequência e taxa de transmissão	31
2.4	COMUNICAÇÃO SERIAL POR <i>UART</i>	31
2.5	COMUNICAÇÃO POR LUZ VISÍVEL	33
2.5.1	Arquitetura da VLC	33
2.6	MICROCONTROLADORES	34
2.6.0.1	Pinos GPIO	34
2.6.0.2	Entrada e saída analógica	34
2.6.0.3	Interrupções	35
2.7	ARDUINO	35
2.7.1	Arduino Mega 2560 Rev3	35
2.7.1.1	Microcontrolador ATmega2560	36
2.7.1.1.1	<i>Portas de entrada e saída (I/O)</i>	36
2.7.1.1.2	<i>Interrupções externas</i>	36
2.7.1.1.3	<i>Interrupções por TIMER/COUNTER de 16 bits</i>	37
2.7.2	Arduino Nano	38
2.7.2.1	Microcontrolador ATmega328p	39
2.7.2.1.1	<i>Portas de entrada e saída (I/O)</i>	39
2.7.2.1.2	<i>Interrupções externas</i>	39
2.7.2.1.3	<i>Interrupções por TIMER/COUNTER de 16 bits</i>	39
2.7.2.2	Módulo TEMT6000	40
3	DESENVOLVIMENTO	41

3.1	DESCRIÇÃO GERAL DO PROJETO	41
3.1.1	Descrição geral do transmissor	41
3.1.2	Descrição geral do receptor	42
3.2	DEFINIÇÕES PARA IMPLEMENTAÇÃO	43
3.2.1	Definição do Pacote de dados	43
3.2.2	Rotina do transmissor	43
3.2.3	Rotina do receptor	45
3.3	DESCRIÇÃO DETALHADA DO TRANSMISSOR	46
3.3.1	Hardware do transmissor	46
3.3.2	Algoritmo do transmissor	47
3.3.2.1	Considerações iniciais, macros e variáveis globais	47
3.3.2.2	Inicialização de variáveis e configuração de registradores	50
3.3.2.3	Rotina <i>loop()</i>	50
3.3.2.4	Rotina de serviço de interrupção	51
3.4	DESCRIÇÃO DETALHADA DO RECEPTOR	51
3.4.1	Hardware Receptor	52
3.4.2	Algoritmo do receptor	56
3.4.2.1	Considerações iniciais, macros e variáveis globais	56
3.4.2.2	Inicialização de variáveis e configuração de registradores	58
3.4.2.3	Rotina <i>loop()</i>	58
3.4.2.4	Rotina de serviço de interrupção	60
3.5	COLETA, COMPILAÇÃO E ANÁLISE DE DADOS	61
3.5.1	Método de coleta de dados	61
3.5.2	Método de análise dos dados	63
3.5.3	Compilação dos dados	64
3.5.4	Função <i>generateJsons()</i>	65
3.5.5	Função <i>compileJsons()</i>	67
3.5.6	Função <i>openJson()</i>	67
3.5.7	Função <i>deleteJsons()</i>	68
4	RESULTADOS E DISCUSSÕES	69
4.1	DIVISOR DE TENSÃO COM $R_1 = 1 \text{ k}\Omega$ E $R_2 = 1 \text{ k}\Omega$	69
4.1.1	Carga de dados de 100 bytes	69
4.1.2	Carga de dados de 500 bytes	72
4.1.3	Carga de dados de 1000 bytes	74
4.2	DIVISOR DE TENSÃO COM $R_1 = 10 \text{ k}\Omega$ E $R_2 = 1 \text{k}\Omega$	76

4.2.1	Carga de dados de 100 <i>bytes</i>	76
4.2.2	Carga de dados de 500 <i>bytes</i>	78
4.2.3	Carga de dados de 1000 <i>bytes</i>	80
4.3	DIVISOR DE TENSÃO COM $R_1 = 50 \text{ k}\Omega$ E $R_2 = 1 \text{ k}\Omega$	82
4.3.1	Carga de dados de 100 <i>bytes</i>	82
4.3.2	Carga de dados de 500 <i>bytes</i>	84
4.3.3	Carga de dados de 1000 <i>bytes</i>	86
4.4	DISCUSSÃO SOBRE OS RESULTADOS	88
4.5	COMPARAÇÃO COM OUTROS TRABALHOS	89
5	CONCLUSÃO E CONSIDERAÇÕES FINAIS	91
5.1	PROBLEMAS ENCONTRADOS	91
5.2	SUGESTÕES PARA TRABALHOS FUTUROS	92
	REFERÊNCIAS	93
	APÊNDICES	97
	APÊNDICE A – MATERIAIS	99
A.1	LISTA COMPLETA DE MATERIAIS	99
A.2	MATERIAIS DO TRANSMISSOR	100
A.3	MATERIAIS DO RECEPTOR	100
	APÊNDICE B – CÓDIGOS EXEMPLO	101
B.1	EXEMPLO 01	101
B.2	EXEMPLO 02	102
	APÊNDICE C – <i>STRINGS</i>	105
C.1	<i>STRING</i> DE 100 CARACTERES (100 <i>BYTES</i>)	105
C.2	<i>STRING</i> DE 500 CARACTERES (500 <i>BYTES</i>)	105
C.3	<i>STRING</i> DE 1000 CARACTERES (1000 <i>BYTES</i>)	105
	APÊNDICE D – CÓDIGO DO TRANSMISSOR	107
D.1	PARTE 01	107
D.2	PARTE 02	108
D.3	PARTE 03	109
D.4	PARTE 04	110
	APÊNDICE E – CÓDIGO DO RECEPTOR	111
E.1	PARTE 01	111
E.2	PARTE 02	111
E.3	PARTE 03	112

E.4	PARTE 04	113
	APÊNDICE F – CÓDIGO DE COMPILAÇÃO E BUSCA DOS DA- DOS GERADOS	115
F.1	CÓDIGO FONTE COMPLETO	115
F.2	CÓDIGO DE EXEMPLO DE RECUPERAÇÃO DE DADOS SIMPLES	122
F.3	CÓDIGO DE EXEMPLO DE RECUPERAÇÃO DE DADOS DETA- LHADO	122

1 INTRODUÇÃO

Neste capítulo será apresentada uma contextualização acerca da *Visible Light Communication (VLC)*, em português Comunicação por Luz Visível, os objetivos do projeto e a organização deste trabalho.

1.1 CONTEXTUALIZAÇÃO

A tendência mundial, com relação ao avanço tecnológico de dispositivos eletrônicos, é a criação de dispositivos que estabeleçam algum tipo de comunicação com a Internet. Como exemplo disso, temos o avanço da tecnologia da *Internet of Things (IoT)*, em português, Internet das Coisas, através da qual cada vez mais os diversos componentes eletrônicos estabelecem algum meio de comunicação via Internet. Estima-se que em 2025 haja cerca de 39 bilhões de dispositivos conectados à Internet e, desconsiderando a IoT industrial, os dispositivos cabeados assumem uma pequena parcela da estimativa ([HELP NET SECURITY, 2019](#)).

O método mais utilizado de comunicação sem fio (*wireless*) é via radiofrequências, como por exemplo *Wireless Fidelity (WiFi)*, em português Fidelidade Sem Fio, e 4G. Com o aumento da quantidade de dispositivos utilizando essas frequências de rádio, cada vez mais há a requisição de bandas do espectro para a realização da comunicação. Isso leva a um possível problema: a saturação da utilização da banda de radiofrequências ([MATHEUS, L. E. M. et al., 2019](#)). Um caso mais recente que envolveu esse problema de saturação no Brasil foi o fim da transmissão analógica para canais de televisão, uma medida tomada para liberar uma faixa do espectro para destiná-la à comunicação de rede de Internet via 4G. Restringindo ao WiFi, há a crise do espectro do WiFi ([ALDERFER, 2013](#)), em que, devido ao grande aumento de dispositivos conectados às bandas de WiFi, a saturação das bandas para essa tecnologia também é bastante importante.

Sabendo disso, uma possível tecnologia para solucionar os problemas que ocorrem com as radiofrequências é a VLC. Trata-se de uma tecnologia que usa o espectro visível do espectro eletromagnético para realização da comunicação. Uma motivação para utilizar tal tecnologia é a não interferência nas ondas de rádio, bem como um provável menor custo de produção da infraestrutura, já que é possível realizá-la com a utilização de LEDs e receptores, adequados a cada situação. Além disso, a velocidade de transmissão de dados por meio da luz permite atingir valores maiores, quando comparado a uma transmissão que utiliza radiofrequências ([KHAN, 2017](#)).

Tal sistema de VLC é mais simples de ser implementado em ambientes internos, sem interferência de luz externa, podendo ser um complemento à tecnologia de radiofrequênci. Um fator que limitava anteriormente a comunicação por luz eram os tipos de lâmpadas, mas com o aumento de dispositivos com LED, tornou-se uma área de pesquisa/aplicação mais viável.

A fim de realizar um sistema de comunicação por meio de luz visível, esse projeto tem como objetivo implementar a tecnologia de VLC junto à plataforma de hardware Arduino e similares. Apesar dessa plataforma não ter um custo relativamente baixo, é uma das plataformas de prototipagem mais acessíveis e de grande acervo online, além de possibilitar a utilização de periféricos de baixo custo, para gerar um produto final eficiente na transmissão de dados e de custo acessível. Com a implementação realizada, pôde-se então fazer as análises acerca de possíveis variáveis que possam interferir na qualidade do sinal, causando falhas na transmissão. Ter uma noção sobre as variáveis que geram impactos negativos, gerando perda de dados ao longo da transmissão, permite que projetos futuros tenham conhecimento em relação às possíveis complicações ao longo do desenvolvimento e suas possíveis causas, bem como um primeiro passo para o estudo sobre possíveis soluções.

1.2 TRABALHOS RELACIONADOS E APLICAÇÕES

Um dos aspectos mais importantes e vantajosos da VLC são as altas frequência do espectro eletromagnético utilizadas na comunicação, permitindo uma alta taxa de transferência de dados ([MATHEUS, L. E. M. et al., 2017](#)). Além disso, as bandas não são especificadas para cada tipo de aplicação na VLC e apresentam alto grau de segurança em ambientes internos, já que a luz não atravessa determinadas superfícies. Com isso, uma aplicação para transmissão de dados subaquáticas com ondas acústicas e ondas ópticas em conjunto torna-se possível ([KAUSHAL; KADDOUM, 2016](#)). As transmissões de dados feitas com um canal de comunicação formado pela água são realizadas através do uso de ondas de radiofrequênci, ondas acústicas e ondas ópticas. Ondas acústicas tem como vantagem a transmissão por longas distâncias, na casa de quilômetros, mas possuem uma velocidade de propagação baixa, gerando uma taxa de transmissão na casa dos kbps ([KAUSHAL; KADDOUM, 2016](#)). Além disso, necessita de dezenas de Watts para realizar a transmissão e possuem um largura de banda na faixa de kHz. Ondas de radiofrequênci e ópticas atingem velocidades de $2.255 \cdot 10^8$ m/s e chegam a uma largura de banda na faixa MHz, sendo que atingem Mbps e Gbps de taxa de transmissão de dados respectivamente. Porém, dentro da

água, radiofrequências ficam limitadas a distâncias de dezenas de metros (QURESHI *et al.*, 2016), enquanto as ópticas atingem até 100m (KAUSHAL; KADDOUM, 2016). Além disso, a comunicação óptica consome poucos Watts para funcionar. Tais métricas sofrem variação de acordo com aspectos da água, como temperatura, salinidade, pressão, condutividade, permissividade, absorção/espalhamento, turbulência e matéria orgânica.

Há ainda a possibilidade de utilização da técnica de *Multiple-Input-Multiple-Output* (MIMO), em português Múltiplas Entradas e Múltiplas Saídas, em ambientes com iluminação de lâmpadas com vários LEDs (MATHEUS, L. E. M. *et al.*, 2017). Através da VLC, é possível montar uma rede de comunicação entre os objetos fixos (postes de iluminação, semáforos, entre outros) e objetos. Além disso, sistemas de comunicação podem ser projetados para funcionar entre veículos, utilizando faróis e fotodiodos, por exemplo (KHAN, 2017). Como alternativa ao uso do sistema de GPS, que são limitados em ambientes internos devido a interferência física das construções, várias técnicas de localização nesses ambientes já foram estudadas, sendo a VLC a solução para problemas de interferência do sinal de rádio e de nível de precisão do posicionamento (MATHEUS, L. E. M. *et al.*, 2017). Podem também ser mencionados estudos sobre a VLC aplicado em ambientes hospitalares, como a utilização a VLC em áreas sensíveis à ondas eletromagnéticas e implementação de um robôs(KHAN, 2017).

Além disso, na última década, houveram estudos na criação de projetos que integrem a comunicação de radiofrequência, por exemplo o *WiFi*, com a VLC, principalmente para aumentar a eficiência dessa comunicação por luz. O sistema híbrido PLiFi (HU *et al.*, 2016), une WiFi e VLC e tem como objetivo solucionar o problema de *Uplink*, ou seja, a transmissão de dados dos dispositivos “receptores” (*smartphones*, *notebooks*, entre outros) aos “transmissores” (pontos de acesso, no caso, algum emissor de luz visível com implementação da tecnologia). Estuda-se também muito a integração com 5G. Outra linha de pesquisa, visa a comunicação utilizando a integração dos *smartphones* com o VLC via suas câmera e seus LEDs externos (MATHEUS, L. E. M. *et al.*, 2017). A NASA lançou em 2013 um projeto com implementação de VLC, no caso OCW, em que a espaçonave enviava dados para Terra através desse sistema, sendo mais rápido que a transmissão por rádio (MATHEUS, L. E. M. *et al.*, 2017). Na área acadêmica, foi desenvolvido uma plataforma (OpenVLC) como tentativa de padronizar os estudos da tecnologia (MATHEUS, L. E. M. *et al.*, 2017).

Focando em trabalhos que utilizam VLC e Arduino, pode ser mencionado (SA-

[ADSAIF0333, 2020](#)) que realiza uma comunicação simples através de um LED e LDR. Em outra implementação, é utilizado não um LED, mas sim um laser ([FAHAD, 2020](#)). Em outro projeto, a comunicação é estabelecida com o uso de LEDs e fototransistores ([HAMED; ODEH; AFANEH, 2015](#)). Por fim, há também uma aplicação utilizada na transferência de dados de áudio ([ELECTRONICS WORKSHOPS, 2020](#)).

1.3 OBJETIVOS

Nessa seção, estão descritos o objetivo geral e os objetivos específicos deste trabalho.

1.3.1 Objetivo Geral

O objetivo geral do trabalho é implementar um sistema que estabeleça a transferência de dados entre dispositivos por meio da utilização da técnica de Comunicação por Luz Visível, de forma otimizada, utilizando para isso plataformas de prototipagem, como Arduino e similares, e a partir disso analisar variáveis que interfiram no índice de erro de transmissão.

1.3.2 Objetivos Específicos

Como objetivos específicos, têm-se:

- Definir os componentes necessários para realizar a comunicação por luz visível;
- Implementar uma comunicação por luz visível, com os componentes definidos, visando a maior taxa de transferência de dados;
- Definição dos dados base para transmissão;
- Validação da transmissão de dados entre plataformas em um cenário ideal;
- Após validação inicial da comunicação, realizar transmissões variando determinados parâmetros do sistema, como por exemplo, a distância entre o receptor e o transmissor, dentre outros fatores. Todas as transmissões devem ser salvas em arquivos;
- Elaborar um algoritmo para compilação de dados, com base nas transmissões coletadas;
- Analisar os dados compilados.

1.4 ORGANIZAÇÃO DA MONOGRAFIA

Esse texto apresenta no [Capítulo 2](#) uma fundamentação teórica de ondas eletrromagnéticas, transmissão de dados e comunicação por luz visível, bem como apresentação das plataformas de desenvolvimento e periféricos utilizados.

Já no [Capítulo 3](#), há um detalhamento sobre o desenvolvimento do projeto, descrevendo as implementações do *hardware* e do algoritmo, bem como metodologias utilizadas e métodos de análise de dados coletados.

No [Capítulo 4](#) há a apresentação dos resultados compilados a partir do projeto realizado, bem como as discussões pertinentes.

Por fim, no [Capítulo 5](#), há um levantamento sobre as conclusões e considerações obtidas acerca do projeto, como pontos problemáticos encontrados e possíveis futuras implementações e modificações.

2 FUNDAMENTAÇÃO TEÓRICA

Neste Capítulo é realizado um levantamento teórico sobre ondas eletromagnéticas e comunicações, focando na comunicação por luz visível. Além disso, há uma fundamentação com relação aos componentes de *hardware* e periféricos utilizados.

2.1 ONDAS ELETROMAGNÉTICAS

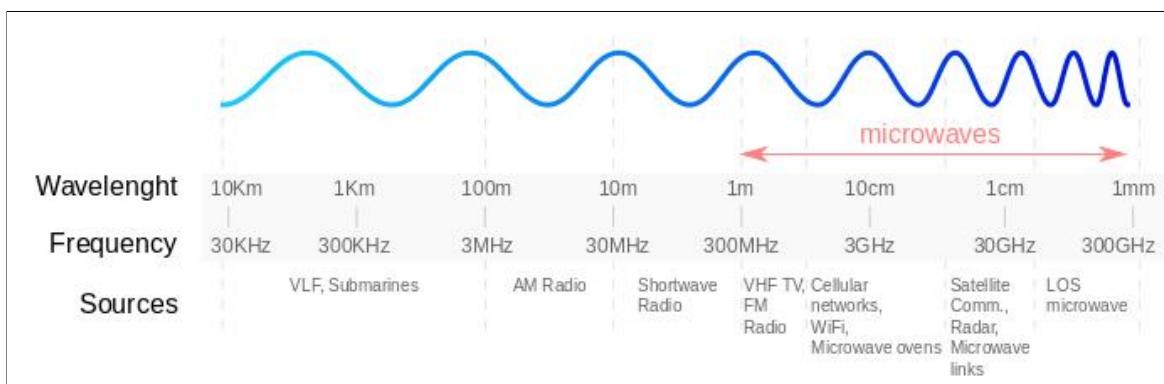
As ondas eletromagnéticas são formadas por um campo elétrico e um campo magnético, sendo esses perpendiculares entre si e, em relação à direção de propagação da onda, transversais. Dá-se o nome de espectro eletromagnético, a faixa de frequências (ou comprimento de onda) da radiação eletromagnética e, dentro dessa faixa, são classificadas regiões de acordo com características/aplicação, por exemplo, faixa de luz visível, de rádio, dentre outras, sendo as regiões do espectro não perfeitamente definidas ([TIPLER; MOSCA, 2009](#)).

Na [Seção 2.1.1](#) e na [Seção 2.1.2](#), são discutidas as classificações do espectro eletromagnético mais importantes para esse trabalho, a saber, as ondas de rádio e luz visível.

2.1.1 Ondas de Rádio

As frequências de rádio compreendem o intervalo do espectro eletromagnético entre aproximadamente 3 kHz a 300 GHz, com comprimento de onda variando aproximadamente entre 100.000 km até 1 mm, sendo esse dividido em diferentes bandas de acordo com suas respectivas propriedades ([UGWEJE, 2004](#)). Por exemplo, bandas de frequência extremamente baixa (menores que 30 kHz) são utilizadas em navegação. Bandas de média frequência (entre 0.3MHz e 3 MHz) são utilizadas na comunicação de rádio AM. As frequências super altas (entre 3 GHz e 30GHz), parte da faixa que compreende as microondas, são utilizadas em satélites ([UGWEJE, 2004](#)). Como é possível observar, pelos exemplos e pela [Figura 1](#), as ondas de rádio são bastante empregadas para a comunicação sem fio.

Figura 1 – Radiofrequências e aplicações



Fonte: *Radio Waves* (YATEBTS, c2019).

Porém, com o emprego extensivo das ondas de rádio nas telecomunicações, surgem diversos problemas de interferências entre diversas aplicações do cotidiano como, por exemplo, a comunicação da aviação comercial e telefones celular. Além disso, as ondas de rádio apresentam limitações de velocidade e vulnerabilidades de segurança, por questões físicas. Outro ponto questionável é a interação das ondas de rádio com a saúde humana, em que pesquisas não se demonstram conclusivas para potenciais problemas de saúde devido a interação humana com comunicações que utilizam a radiofrequência (HABASH *et al.*, 2009)

2.1.2 Luz Visível

O espectro eletromagnético referente à luz visível compreende os comprimentos de onda de aproximadamente 380nm até 740nm, com a faixa de frequência variando de aproximadamente 405 THz à 790 THz. Na Figura 2 é possível observar as informações mencionadas, bem como a cor correspondente a cada faixa de frequência. Por exemplo, a faixa que compreende a cor vermelha (Red na Figura 2) apresenta frequência mínima de 405 THz e máxima de 480 THz, sendo o seu comprimento mínimo de 625 nm e máximo de 740 nm. Demais cores podem ser observadas tanto pela cor da linha na imagem, como pelo nome da cor em inglês na coluna mais à esquerda. De cima para baixo, traduzindo o nome das cores, temos vermelho, laranja, amarelo, verde, azul, índigo (anil) e violeta. Já as duas outras colunas, “Wavelength” e “Frequency”, são traduzidas para comprimento de onda e frequência . Demais cores podem ser formadas através de uma composição das cores da Figura 2, variando a intensidade de emissão.

Figura 2 – Comprimentos de onda e bandas de frequência da luz visível

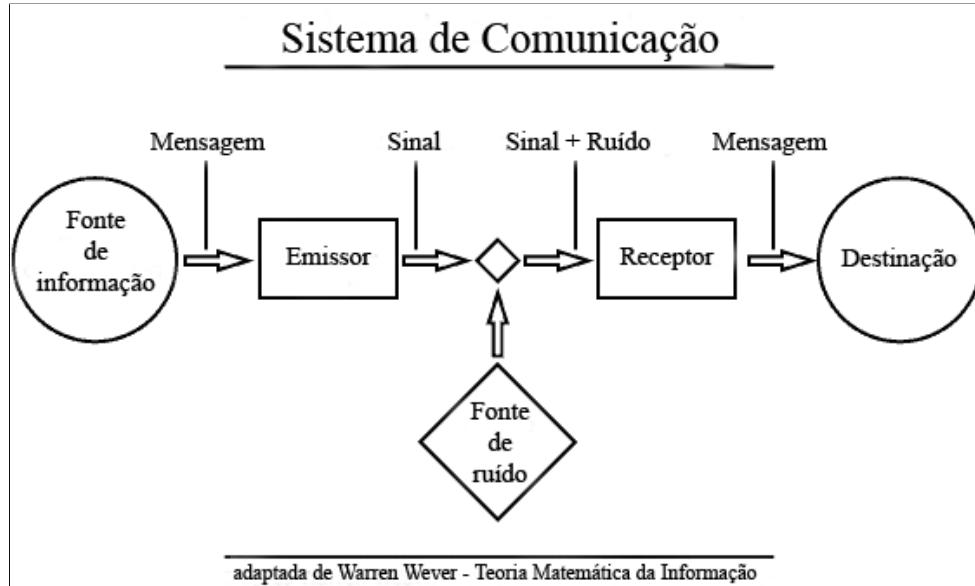
	Wavelength	Frequency
Red	~ 625 – 740 nm	~ 480 – 405 THz
Orange	~ 590 – 625 nm	~ 510 – 480 THz
Yellow	~ 565 – 590 nm	~ 530 – 510 THz
Green	~ 520 – 565 nm	~ 580 – 530 THz
Blue	~ 445 – 520 nm	~ 675 – 580 THz
Indigo	~ 425 – 445 nm	~ 700 – 675 THz
Violet	~ 380 – 425 nm	~ 790 – 700 THz

Fonte: *The Electromagnetic spectrum* ([UNIVERSITY OF BERGEN, 2021](#)).

2.2 SISTEMAS DE COMUNICAÇÃO

Um sistema de comunicação é utilizado para transferência e recepção de informações. Apresenta como configuração base um módulo transmissor, um canal de transmissão e um módulo receptor, conforme mostrado na [Figura 3](#).

Figura 3 – Esquema básico de um sistema de comunicação



Fonte: Cognição e redes abertas: a informação interativa como coração dos sistemas inteligentes ([BRENNAND; BRENNAND, 2007](#)).

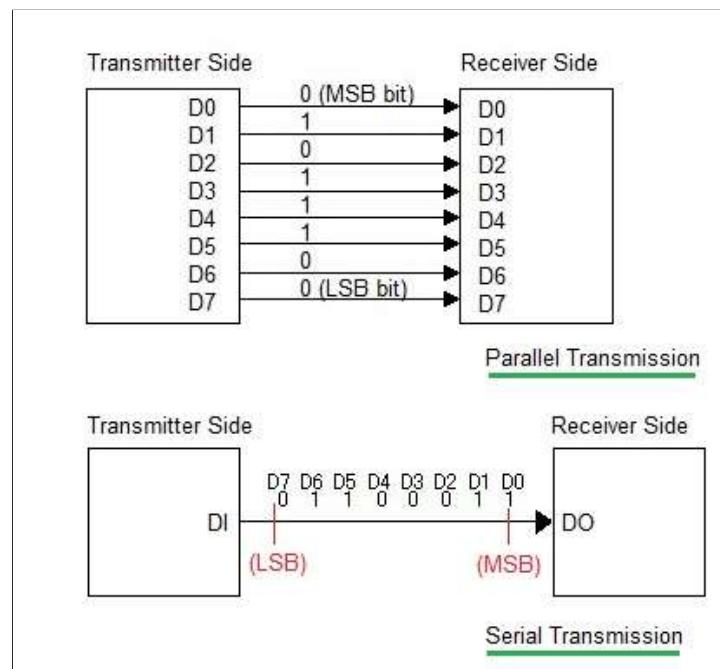
Com base nas partes mostradas na [Figura 3](#) e de forma resumida, o transmissor, ou emissor, realiza a captação de um sinal e modifica esse sinal para enviar de forma eficiente a informação por meio de um canal de comunicação ([LATHI, 2012](#)). O canal de comunicação (losango na [Figura 3](#)) é o meio de transmissão por onde o sinal

modificado pelo transmissor é transmitido. Por sua vez, canal de comunicação está sujeito a diversas fontes de interferências, sendo estas capazes de inserir ruídos (LATHI, 2012). Por fim, o sinal propagado ao longo do meio de comunicação é recebido pelo receptor, sendo esse módulo o responsável por recuperar o sinal e aplicar técnicas de remoção/mitigação de ruídos (LATHI, 2012).

2.3 COMUNICAÇÃO DIGITAL

A comunicação digital é a troca de informação por meio de sinais digitais. Tal comunicação pode ser estabelecida de modo serial (sequencial) ou de modo paralelo, como mostrado na Figura 4

Figura 4 – Esquemático entre comunicação serial e paralela



Fonte: Advantages of Serial Interface | disadvantages of Serial Interface ([RF WIRELLES WORLD](#), c2012).

Na transferência de dados paralela, vários dados podem ser enviados de forma paralela em um único instante de tempo (COWLEY, 2007), como mostrado na Figura 4. Devido ao envio simultâneo de dados, o mecanismo de recebimento dos dados pode ocasionar em informações errôneas, caso esse não seja devidamente implementado.

Em oposição à transmissão paralela, a transmissão de dados serial, também mostrada na Figura 4, realiza o envio sequencial dos dados em um único meio de transmissão de forma síncrona ou assíncrona (COWLEY, 2007). Pelo envio ser realizado de modo sequencial, os dados devem ser previamente ordenados, de modo que o re-

cebimento sequencial de dados, pelo receptor, resulte de fato na informação desejada. Por possuir apenas um canal de comunicação, realizando o envio de um único dado por instante de tempo, a comunicação serial tende a apresentar menores taxas de transmissão quando comparada à transmissão paralela. Devido à baixa complexidade de implementação, em comparação à transmissão paralela, os produtos finais tendem a apresentar um menor custo (LATHI, 2012).

Com relação à sincronia de transmissões seriais, sistemas assíncronos não apresentam um sinal de sincronização comum, como por exemplo um relógio. Assim, a sincronia é feita por meio da inserção de dados (*bits*) de sincronização, indicando fundamentalmente o ponto de início e de fim de um dado efetivo (COWLEY, 2007). Tal inserção de *bits* extra para sincronização pode ocasionar comunicações mais lentas, em comparação à uma comunicação síncrona. Em contrapartida, a comunicação síncrona prevê uma sincronia entre os módulos de transmissão e recepção, permitindo uma taxa de envio de dados constante (COWLEY, 2007), sendo os *bits* de sincronização desnecessários.

Além disso, uma comunicação pode ser classificada com relação ao sentido do fluxo dos dados em uma transmissão. Uma comunicação que prevê o envio de dados em um único sentido, do transmissor para o receptor, é nomeada como *Simplex* (COWLEY, 2007). Porém, se a transferência é feita de forma bilateral alternada, ou seja, há o envio de dados tanto do transmissor para o receptor quanto do receptor para o transmissor, mas em tempos diferentes, tem-se uma comunicação *Half-Duplex* (COWLEY, 2007). Por fim, quanto há a transferência de dados bilateral e essa transferência pode ocorrer de forma simultânea, configura-se um comunicação *Duplex* (COWLEY, 2007).

2.3.1 Frequência e taxa de transmissão

Através da frequência utilizada em um sistema de comunicação digital, é possível também obter a taxa de transmissão *debits* (*bit rate*). O *bit rate* pode ser obtido através da seguinte fórmula:

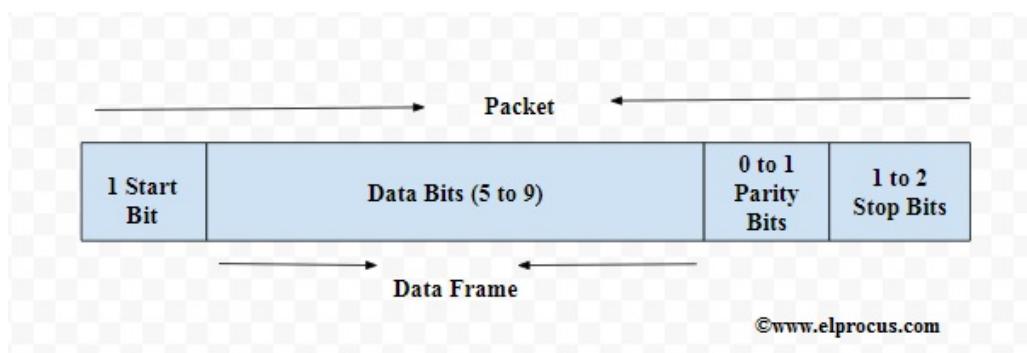
$$\text{Bit rate} = \text{Bits por amostragem} \cdot \text{Frequência de amostragem} \quad (1)$$

2.4 COMUNICAÇÃO SERIAL POR UART

Um sistema que aplica comunicação serial assíncrona é o *Universal Asynchronous Receiver/Transmitter (UART)*, em português, “Transmissor/Receptor Universal

Assíncrono”, sendo um módulo *UART* formado, basicamente, por um módulo receptor, um módulo transmissor e um gerador de *baud rate*. Além de ser assíncrono, pode adotar o fluxo de comunicação *Duplex*. A comunicação é realizada entre interfaces *UARTs*, tal que os módulos de transmissão e recepção implementam mecanismos de conversão de dados paralelos para sequenciais bem como o inverso, respectivamente, para realizar um envio de dados e frente ao recebimento de dados. O gerador de *baud rate* é responsável por gerar sinais de *clock* e controlar a transferência de dados (LADDHA; THAKARE, 2013). Por não apresentar um meio sincronizador entre os módulos, a sincronização é feita através da inserção de *bits* de sincronização nos pacotes de informação, como mencionado na Seção 2.3. Dessa forma, a implementação apresenta fundamentalmente um pacote contendo 1 *bit* de início, *bits* de dados, 1 ou nenhum *bit* de paridade e de 1 até 2 *bits* de parada (LADDHA; THAKARE, 2013), como mostrado na Figura 5.

Figura 5 – Pacote de dados enviados pela comunicação por *UART*



Fonte: UART Communication : Block Diagram and Its Applications ([ELPROCUS](#), s.d.).

Em uma comunicação utilizando *UART*, a saída do transmissor é mantida em nível lógico alto enquanto não há dado a ser transmitido. Assim, o nível lógico de saída do transmissor é alterado para baixo (0) por um ciclo frente a um *bit* de início e o receptor, ao detectar esse nível lógico baixo, pode realizar as etapas necessárias para captar os dados efetivos da informação até a detecção de um *bit* de parada. Os *bits* de dados efetivos tendem a ser inseridos do menos significativo para o mais significativo nas implementações. Inverso ao *bit* de início, o *bit* de parada é representado por um ciclo de *clock* em nível lógico alto (1). Com relação ao *bit* de paridade, que pode ser ausente, esse representa a quantidade de *bits* com nível lógico alto na mensagem, sendo que uma quantidade par altera o *bit* paridade para um nível lógico baixo e uma quantidade ímpar para um nível lógico alto. Com isso, o módulo receptor pode gerar um valor de paridade local, com base nos dados recebidos, e comparar com a paridade

recebida, indicando problemas no pacote de dados recebido caso as paridades não sejam iguais.

2.5 COMUNICAÇÃO POR LUZ VISÍVEL

A Comunicação por Luz Visível trata-se da modulação das ondas eletromagnéticas compreendidas na faixa do espectro de luz visível (380 nm a 740 nm) para realizar a transmissão de dados. Porém, um ponto importante é realizar essa transmissão de modo que não seja perceptível e/ou não cause traumas ao olho humano.

Um dos problemas que são solucionados pela utilização dessa tecnologia é a escassez de bandas de rádio frequência, já que a faixa compreendida pela luz visível é cerca de 1.000 vezes maior que a faixa de rádio. Além disso, devido ao tamanho das ondas, como exemplificado na [Seção 2.1](#), o receptor de dados para utilização da VLC deve estar dentro da mesma sala do emissor, aumentando o nível de segurança de transmissão dos dados, sendo que a transmissão de rádio não necessita de tal restrição. Outro ponto de vantagem à VLC é que os emissores de luz poderão ser também utilizados para iluminar, além de transmitir dados, ou seja, potencialmente diminuirão os custos de instalação de infraestrutura ([KHAN, 2017](#)). Por fim, não existe interferência entre o espectro visível e o de rádio, o que possibilita o uso dos dois sistemas de forma simultânea.

Apesar disso, necessita-se ressaltar alguns pontos negativos com relação a comunicação por luz visível, como a interferência com a luz ambiente, interferência entre dispositivos que utilizam tal tecnologia e a integração com os meios de comunicação já existentes. Como forma de padronizar a transmissão por VLC, pode ser citado o padrão o IEEE 802.15.7, o qual impõe regras para diferentes taxas de transmissão, quantidade de fontes de emissão de dados, bem como para níveis de alteração de luminosidade que evite o efeito de cintilação ([RAJAGOPAL; ROBERTS; LIM, 2012](#)).

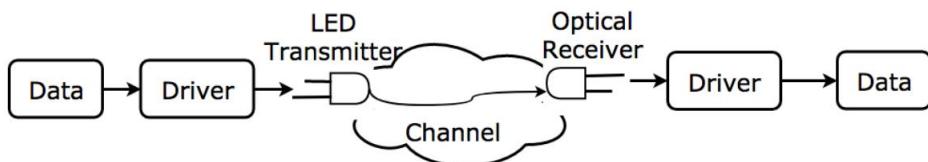
A comunicação óptica sem fio é bastante utilizada através do infravermelho. Porém, somente na primeira década dos anos 2000 foram realizados experimentos para estabelecer comunicação, em ambientes internos, com LEDs brancos. Tal tecnologia ganhou ainda mais visibilidade após a apresentação da tecnologia de *Light Fidelity (LiFi)* ([HAAS et al., 2016](#)).

2.5.1 Arquitetura da VLC

A arquitetura base de uma comunicação VLC consiste em um módulo transmissor com um modulador de sinal e uma fonte emissora de luz, a qual fará o envio do

signal de acordo com a intensidade de luz. Essa luz é enviada por um canal de comunicação, onde podem haver interferências de outras fontes de luz. Por fim, um módulo receptor, contendo um receptor de intensidade de luz e um demodulador, recebe o sinal e o converte para o dado efetivo. Um exemplo é mostrado na [Figura 6](#).

Figura 6 – Arquitetura base da VLC



Fonte: The internet of light: Impact of colors in LED-to-LED visible light communication systems ([MATHEUS, L. et al., 2019](#)).

2.6 MICROCONTROLADORES

Os microcontroladores são pequenos computadores que contém uma unidade central de processamento (do inglês, [*Central Processing Unit \(CPU\)*](#)), uma memória e periféricos de entrada e saída ([GRIDLING; WEISS, 2007](#)). Microcontroladores são especializados na execução de tarefas específicas, normalmente de Sistemas Embarados.

Nas próximas subseções, serão melhor detalhadas informações acerca de um microcontrolador, como os pinos GPIO, a entrada e saída analógica e as interrupções.

2.6.0.1 Pinos GPIO

Pinos de entrada e saída de propósito geral, em inglês, [*General Purpose Input Output \(GPIO\)*](#), trabalham com valores lógicos “alto” e “baixo” com base em uma tensão de referência estabelecida no microcontrolador. Através de registradores, podemos alterar o estado de um pino *I/O* (nível alto ou nível baixo), definir se tal pino é uma entrada ou uma saída de dados e obter o valor em um pino ([GRIDLING; WEISS, 2007](#)).

2.6.0.2 Entrada e saída analógica

Diferentemente das entradas e saídas digitais, as analógicas não trabalham com valores alto e baixo, mas sim com um valor proporcional à tensão de entrada/saída([GRIDLING; WEISS, 2007](#)). Alguns microcontroladores possuem conversores analógicos para digital e digital para analógico.

2.6.0.3 Interrupções

Interrupções são sinais emitidos para a unidade central de processamento indicando que um evento precisa ser tratado de forma imediata. Como exemplos de um evento, temos a alteração do valor de um pino configurado como entrada (interrupção externa) e quando um contador atinge seu valor máximo de contagem (interrupção por tempo). Interrupções podem ser devidamente configuradas através de registradores reservados no microcontrolador.

A interrupção vem como solução para o problema de *polling* ([GRIDLING; WEISS, 2007](#)), ou seja, de monitorar, em várias partes do programa, ou com alta frequência no programa em execução no microcontrolador, o estado atual de um dispositivo. Realizar tal procedimento no microcontrolador pode ser custoso, em termos de desempenho, e causar lentidão na execução. Dessa forma, o programa em execução não consome processamento avaliando várias vezes uma condição. A interrupção é feita por uma rotina de serviço de interrupção, em inglês *Interrupt Service Routine (ISR)*, a qual tem como parâmetro o tipo de interrupção ([GRIDLING; WEISS, 2007](#)).

Frente a um evento de interrupção, uma *flag* é alterada, indicando a interrupção ao processador. É aguardado até que a instrução atual seja finalizada e então iniciada a devida ISR, caso as interrupções estejam habilitadas e tal interrupção esteja habilitada para aquele determinado evento ([GRIDLING; WEISS, 2007](#)).

2.7 ARDUINO

O Arduino é uma plataforma de prototipagem, compreendendo tanto o software quanto o hardware, utilizado por pessoas com diferentes níveis de conhecimento. Além disso, por ser *open source*, é uma plataforma adaptável a diversas soluções e implementações ([ARDUINO, c2021](#)). Algumas das vantagens em se utilizar Arduino são: baixo custo, quando comparado com outras plataformas de microcontroladores; a IDE funciona em diversos sistemas operacionais (Windows, Mac e Linux); a IDE de programação simples e é *open source* ([ARDUINO, c2021](#)).

2.7.1 Arduino Mega 2560 Rev3

O Arduino Mega 2560 Rev3 conta com o microcontrolador de 8bits ATmega2560. Possui 54 pinos de entrada e saída digitais, 16 pinos de entrada analógica e 15 pinos PWM. Possui um *clock* de sistema de 16MHz gerado por um cristal e 3 memórias (8KB

de SRAM, 256KB de FLASH e 4KB de EEPROM) ([ARDUINO, c2022a](#)).

O mapa de pinos do Arduino Mega 2560 Rev3 pode ser consultado na [documentação do Arduino Mega Rev3](#) ([ARDUINO, c2022a](#)).

2.7.1.1 Microcontrolador ATmega2560

As informações destacadas nesta seção serão aquelas utilizadas ao longo do desenvolvimento do projeto. As informações aqui presentes, podem ser consultadas no [datasheet do ATmega2560](#) ([ATMEL CORPORATION, 2014](#)).

2.7.1.1.1 Portas de entrada e saída (I/O)

As portas I/O podem ser manipuladas através dos registradores:

- DDRx (*Data Direction Register*): configura se o pino de uma porta será de entrada (valor lógico 0) ou de saída (valor lógico 1);
- PORTx: configura o valor lógico de um pino de uma porta, valor alto (1) ou valor baixo (0) ;
- PINx: guarda o estado lógico de um pino.

Cada um desses registradores possuem 8 bits, sendo que cada um configura uma determinada porta. No ATmega2560, o valor de “x” dos registradores listados acima variam de “A” a “L”.

2.7.1.1.2 Interrupções externas

As interrupções externas podem ser configuradas através dos registradores:

- EICRx: configura o tipo de interrupção externa no pino da porta. A configuração de cada porta ocorre por meio de 2 bits;
- EIMSK: ativa (valor 1) ou desativa (valor 0) interrupções externas em determinada porta. EIMSK[n] manipula a porta com INTn.

O ATmega2560 apresenta EICRA e EICRB, onde cada um configura 4 portas de interrupção externa. Considerando 2bits (mais significativo à esquerda) de configuração de uma porta qualquer com interrupção externa, as configurações possíveis são mostradas no [Quadro 1](#).

Quadro 1 – Configurações feitas via registrador EICRx

Bits	Configuração
00	Gera uma interrupção frente a um valor baixo no pino da porta.
01	Gera uma interrupção frente a uma troca de valor no pino da porta.
10	Gera uma interrupção frente a uma borda de descida entre dois valores no pino da porta.
11	Gera uma interrupção frente a uma borda de subida entre dois valores no pino da porta.

Fonte: [Datasheet ATmega2560 \(ATMEL CORPORATION, 2014\)](#)

2.7.1.1.3 Interrupções por TIMER/COUNTER de 16 bits

As interrupções por *TIMER/COUNTER* de 16 bits (TIMER/COUNTER 1,3,4 e 5) podem ser configuradas pelos bits dos registradores na [Quadro 2](#). Ao longo desse seção de interrupções, os valores de “n” compreendem os números 1, 3, 4 e 5 e os valores de “x” compreendem as letras A, B e C.

Quadro 2 – Configurações feitas via registrador TCCRnx

Bits	Configuração
TCCRnA[7:6]	Configura o modo de comparação de saída do TIMER/COUNTER n no Canal A. Utiliza registradores OCnx.
TCCRnA[5:4]	Configura o modo de comparação de saída do TIMER/COUNTER n no Canal B. Utiliza registradores OCnx.
TCCRnA[3:2]	Configura o modo de comparação de saída do TIMER/COUNTER n no Canal C. Utiliza registradores OCnx.
TCCRnA[1:0]	Junto aos bits TCCRnB[4:3], configura o modo da geração da forma de onda.
TCCRnB[7]	Ativa o filtro de cancelamento de ruído do pino de captura de entrada.
TCCRnB[6]	Seleciona a borda, subida ou descida, do pino de captura de entrada.
TCCRnB[5]	RESERVADO.
TCCRnB[4:3]	Junto aos bits TCCRnA[1:0], configura o modo da geração da forma de onda.
TCCRnB[2:0]	Seleciona a fonte do <i>clock</i> utilizado pelo TIMER/COUNTER ou configura o <i>prescaler</i> para o <i>clock</i> de 16MHz.

Fonte: [Datasheet ATmega2560 \(ATMEL CORPORATION, 2014\)](#)

Os modos de geração de forma de onda podem ser conferidos no *datasheet* do microcontrolador em questão. No projeto que será descrito ao longo da redação, será utilizado apenas a configuração *Clear Timer on Compare Match* (CTC), em que o contador é zerado, reiniciando o valor do registrador TCNTn, quando os valores de TCNTn

(contador) e OCRnx (valor de referência a ser comparado com TCNTn no Canal x com OCRnx) são iguais. Como são contadores de 16 *bits*, os registradores TCNTn são compostos por 2 registradores de 8 *bits*, TCNTnH (TCNTn[15:8]) e TCNTnL(TCNTn[7:0]). O mesmo ocorre com os registradores OCRnx, apresentando OCRnxH e OCRnxL.

O registrador TIMSKn determina se um tipo de interrupção por tempo está ou não habilitada, como no [Quadro 3](#).

Quadro 3 – Configurações feitas via registradores TIMSKn

Bits	Configuração
TIMSKn[7:6]	RESERVADO.
TIMSKn[5]	Ativa (1) ou desativa (0) interrupção por captura de entrada.
TIMSKn[4]	RESERVADO.
TIMSKn[3]	Ativa (1) ou desativa (0) interrupção por comparação no Canal C.
TIMSKn[2]	Ativa (1) ou desativa (0) interrupção por comparação no Canal B.
TIMSKn[1]	Ativa (1) ou desativa (0) interrupção por comparação no Canal A.
TIMSKn[0]	Ativa (1) ou desativa (0) interrupção por estouro do contador.

Fonte: [Datasheet ATmega2560 \(ATMEL CORPORATION, 2014\)](#)

Já o registrador TIFRn guarda os valores das *flags* referente as interrupções. O valor alto indica que uma interrupção foi lançada. As *flags* são zeradas após a execução da rotina de interrupção. As respectivas indicações são mostradas em [Quadro 4](#).

Quadro 4 – Configurações feitas via registradores TIFRn

Bits	Configuração
TIFRn[7:6]	RESERVADO.
TIFRn[5]	Indica interrupção por captura de entrada.
TIFRn[4]	RESERVADO.
TIFRn[3]	Indica interrupção por comparação no Canal C.
TIFRn[2]	Indica interrupção por comparação no Canal B.
TIFRn[1]	Indica interrupção por comparação no Canal A.
TIFRn[0]	Indica interrupção por estouro do contador.

Fonte: [Datasheet ATmega2560 \(ATMEL CORPORATION, 2014\)](#)

Exemplos de código que implementam interrupções para o Arduino Mega 2560 Rev3 podem ser encontrados no [Apêndice B](#).

2.7.2 Arduino Nano

O Arduino Nano conta com o microcontrolador de 8 *bits* ATmega328p. Possui 14 pinos de entrada e saída digitais, 8 pinos de entrada analógica e 6 pinos PWM. Possui

um *clock* de sistema de 16MHz e 3 memórias (2KB de SRAM, 32KB de FLASH e 1KB de EEPROM) ([ARDUINO, c2022b](#)).

O mapa de pinos do Arduino Mega 2560 Rev3 pode ser consultado na [documentação do Arduino Nano](#) ([ARDUINO, c2022b](#)).

2.7.2.1 Microcontrolador ATmega328p

As informações destacadas nesta seção serão aquelas utilizadas ao longo do desenvolvimento do projeto. As informações aqui presentes, e mais informações, podem ser consultadas no [datasheet do ATmega328p](#) ([ATMEL CORPORATION, 2015](#)).

2.7.2.1.1 Portas de entrada e saída (I/O)

As portas I/O podem ser manipuladas através dos registradores mesmos registradores mencionados para o microcontrolador ATmega2560, DDRx, PORTx e PINx. No ATmega328p, o valor de “x” dos registradores listados nesse parágrafo varia de “A” à “D”.

2.7.2.1.2 Interrupções externas

As interrupções externas também podem ser configuradas através dos registradores mencionados para o ATmega2560, EICRx e EIMSK.

O ATmega328p apresenta apenas o registrador EICRA, sendo que configura as únicas 2 portas de interrupção externa. Considerando 2bits (mais significativo à esquerda) de configuração de uma porta qualquer com interrupção externa, as configurações possíveis são idênticas às mostradas no [Quadro 1](#).

Os bits EICRA[7:4] são reservados, enquanto os bits EICRA[3:2] configuram a porta D3 (pino D3) e os bits EICRA[1:0] configuram a porta D2 (pino D2).

2.7.2.1.3 Interrupções por TIMER/COUNTER de 16 bits

As interrupções por TIMER/COUNTER de 16bits (TIMER/COUNTER 1) podem ser configuradas pelos bits dos registradores no [Quadro 2](#), porém, com n=1, já que possui apenas um TIMER/COUNTER de 16 bits. Além disso, diferentemente do microcontrolador anterior, não há o Canal C e, portanto, os bits TCCR1A[3:2] são também reservados.

Ao longo dessa seção de interrupções, os valores de “x” compreendem as letras A e B.

Os modos de geração de forma de onda podem ser conferidos no *datasheet* ([ATMEL CORPORATION, 2014](#)) do microcontrolador em questão. A configuração utilizada no projeto, como mencionado para o microcontrolador anterior, é a *CTC* e segue o mesmo padrão já descrito, mas com valor de $n=1$;

Os registradores TIMSKn e TIFRn seguem os mesmo padrão do [Quadro 3](#) e [Quadro 4](#), assumindo $n=1$, respectivamente, mas com os bits TIMSK1[3] e TIRF1[3] também reservados.

2.7.2.2 Módulo TEMT6000

O módulo TEMT6000 de sensor de luz visível é composto pelo fototransistor TEMT6000 e por um resistor de $10\ k\Omega$. Além disso, possui as trilhas, pinos, para tensão de entrada, aterramento e tensão de saída ([SPARKFUN, \[2003 - 2022\]](#)).

O fototransistor TEMT6000 é sensível ao espectro da luz visível. Possui a onda de pico de sensibilidade (λ_p) de 570 nm, com uma banda de detecção de espectro de 360 nm à 970 nm ([VISHAY, c2022](#)).

3 DESENVOLVIMENTO

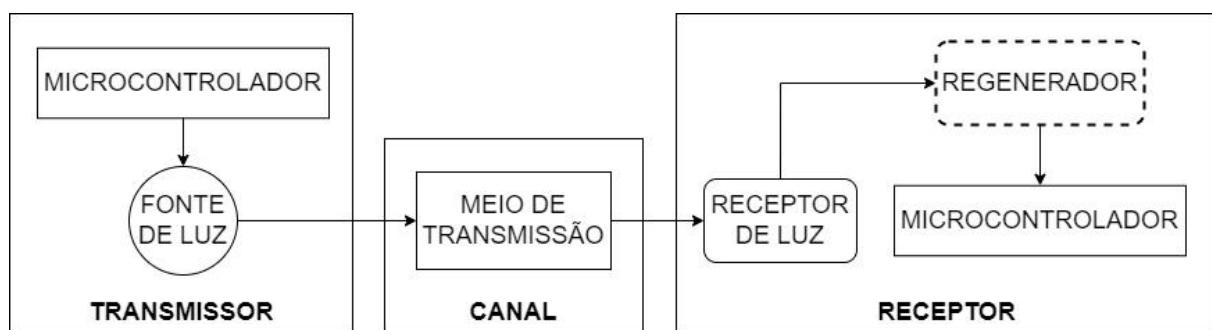
Neste capítulo será detalhado as partes do desenvolvimento do trabalho, partindo do alto nível para o maior detalhamento. Além disso, é descrita a metodologia utilizada no desenvolvimento e para a obtenção e análise dos dados.

A metodologia de desenvolvimento geral do projeto foi feita de forma semelhante à metodologia Scrum, com *reviews* e *plannings* semanal, porém sem as reuniões *daily*.

3.1 DESCRIÇÃO GERAL DO PROJETO

No mais alto nível, representado pela [Figura 7](#), pode-se observar que basicamente é implementada uma comunicação em que o transmissor realiza o envio da informação através de variações da intensidade de luz e o receptor realiza a recepção dessa intensidade

Figura 7 – Diagrama em alto nível do projeto



Fonte: Autor.

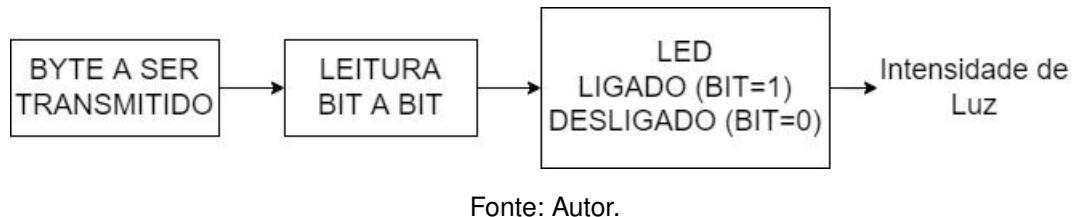
Com essa comunicação estabelecida, é possível analisar variáveis de impacto em uma transmissão de dados realizada por uma comunicação por luz visível.

A lista completa com os materiais utilizados ao longo do desenvolvimento do projeto pode ser encontrada no [Seção A.1](#).

3.1.1 Descrição geral do transmissor

O transmissor faz a leitura dos *bits*, *bit a bit*, de um *byte* de um caractere de uma *string* armazenada na memória, e os envia sequencialmente por um canal de comunicação, nesse caso o ar. De acordo com o valor do *bit*, 0 ou 1, a fonte de luz da [Figura 7](#) (um LED) é desligada ou ligada. O diagrama do transmissor é apresentado na [Figura 8](#).

Figura 8 – Diagrama em alto nível do transmissor

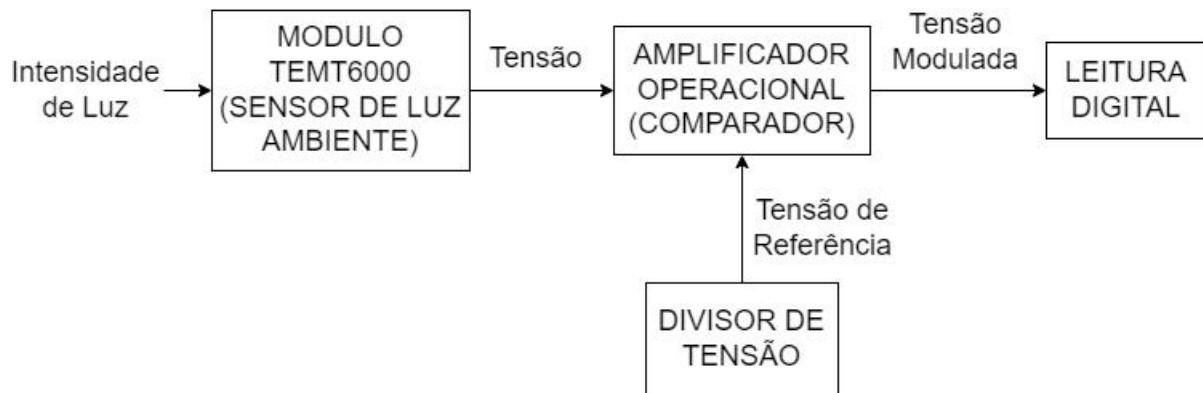


Fonte: Autor.

3.1.2 Descrição geral do receptor

Com relação ao receptor, representado no diagrama da [Figura 9](#), um módulo sensor de luz visível (módulo TEMT6000, que é o Receptor de Luz da [Figura 7](#)), é responsável por capturar a intensidade de luz recebida e gerar uma tensão de saída relacionada a essa intensidade. Quanto mais incidente for a luz sobre o sensor, maior será o valor da saída.

Figura 9 – Diagrama em alto nível do transmissor



Fonte: Autor.

Como a saída do TEMT6000 depende da intensidade de luz recebida, e portanto, é analógica, é necessário realizar uma opção de projeto em relação ao tratamento deste valor. A primeira opção é obter o valor do dado através do conversor analógico/digital do ATmega2560, ao passo que a segunda opção é realizar leituras digitais através de GPIOs, por meio de uma interface de conversão. A primeira opção, apesar de mais simples em termos de hardware, impõe uma forte restrição temporal na leitura do *bit* devido ao tempo de conversão. A segunda opção, apesar de necessitar de uma interface de conversão simples que indique somente se o nível recebido é alto ou baixo, é mais rápida. Portanto, neste projeto, optou-se em obter o dado através de GPIOs por meio de uma interface de conversão de dados na saída do TEMT6000.

Para isso, foi utilizado um amplificador operacional configurado em modo comparador para converter a saída analógica em TEMT6000 em uma saída digital. A tensão

de referência do comparador é obtida a partir de um divisor de tensão, enquanto a tensão a ser comparada é a tensão de saída do módulo sensor de luz visível. Dessa forma, é possível realizar uma leitura digital do sinal analógico proveniente da saída do TEMT6000, já que apresentará apenas duas tensões possíveis. O comparador é o bloco Regenerador da [Figura 7](#).

3.2 DEFINIÇÕES PARA IMPLEMENTAÇÃO

Ao longo dessa seção, são descritas as definições bases feitas para a implementação do *firmware*, no caso, a definição do pacote de dados e das rotinas do transmissor e do receptor.

3.2.1 Definição do Pacote de dados

O pacote de dados definido para o projeto segue o padrão pré-definido pela comunicação *UART*, mais especificamente com 10 *bits*, sendo 1 de início (*START BIT*), 8 de dados efetivos (*DATA BITS*) e 1 de parada (*STOP BIT*). Uma representação gráfica é mostrada na [Figura 10](#). Além disso, o *bit* de início é representado por um valor baixo (0) e o *bit* de parada será representado por um valor alto (1).

Figura 10 – Pacote de dados utilizado no projeto

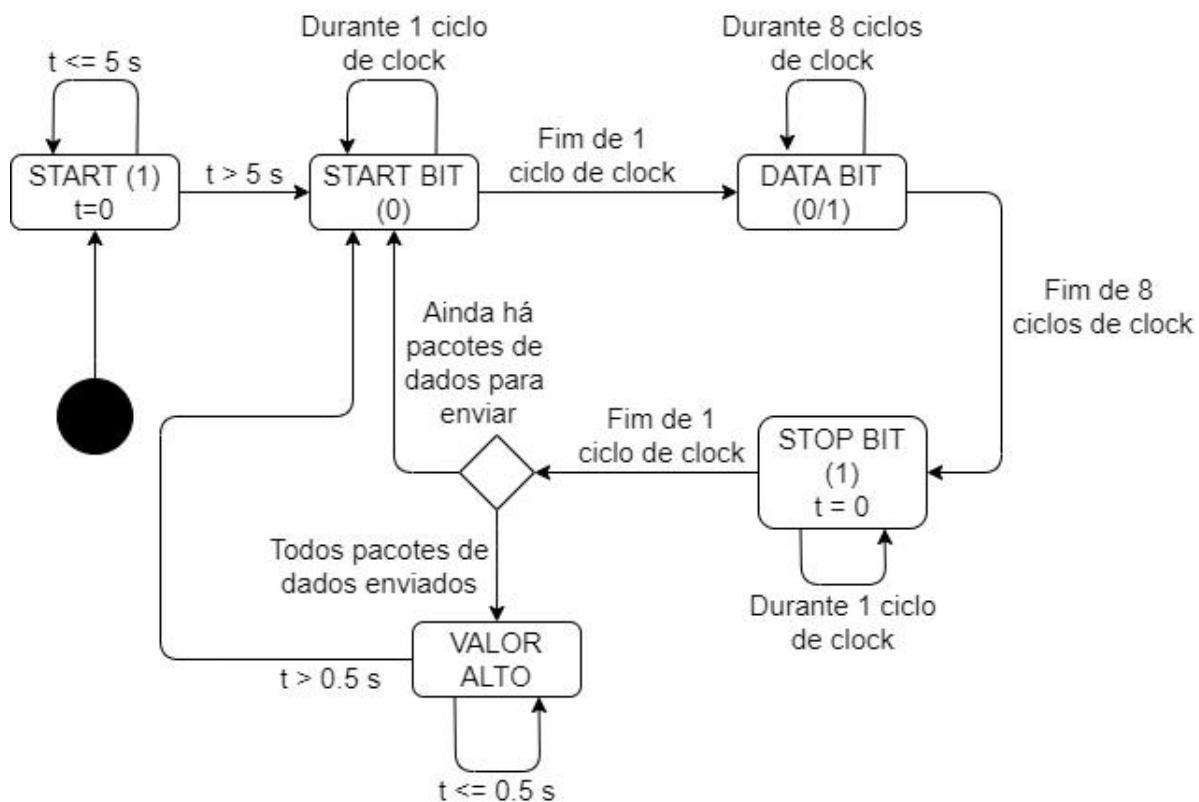
START BIT (1 BIT)	DATA BITS (8 BITS)	STOP BIT (1 BIT)
----------------------	-----------------------	---------------------

Fonte: Autor.

3.2.2 Rotina do transmissor

Um diagrama de máquina de estado para o módulo transmissor é apresentado na [Figura 11](#). No programa, todos os tempos são calculados por meio de interrupções por comparação do *timer* [Seção 2.7.1.1.3](#). Com isso, é possível atribuir o mesmo intervalo de tempo a cada *bit* a ser enviado, permitindo o envio *bit* a *bit*.

Figura 11 – Máquina de estados para o módulo transmissor



Fonte: Autor.

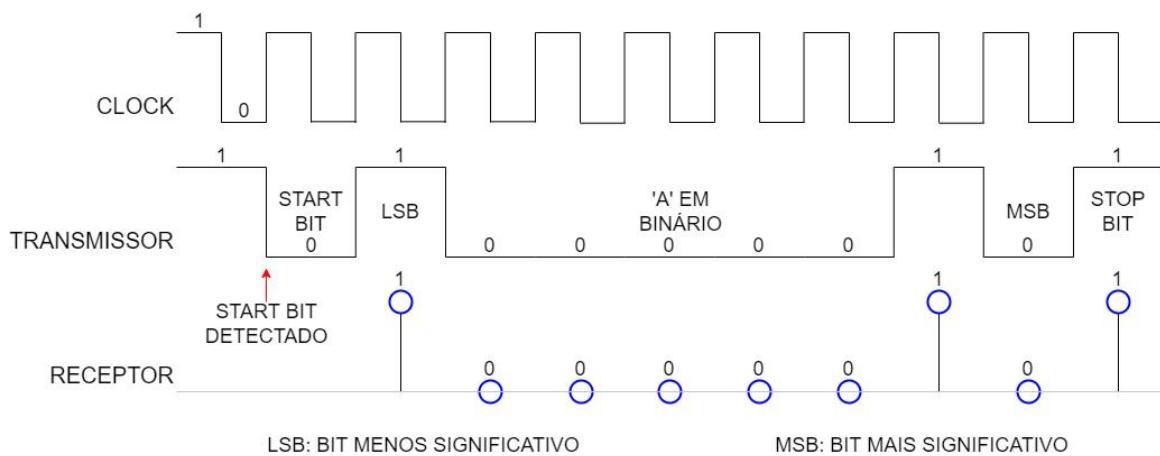
O transmissor segue a seguinte ordem de execução:

1. Inicializa o sistema;
2. Faz o envio de valor alto (1) por 5 s;
3. Faz o envio do pacote de dados em ordem:
 - a) Faz o envio do *bit* de início em 1 ciclo de *clock*;
 - b) Faz o envio dos *bits* de dados efetivos, do menos significativo para o mais significativo, em 8 ciclos de *clock* (1 *bit* por ciclo);
 - c) Faz o envio do *bit* de parada de parada em 1 ciclo de *clock*;
4. Repete o passo 3 enquanto houverem pacotes de dados a serem enviados (quantidade de pacotes enviados é menor ou igual ao tamanho da *string* de envio). A quantidade de pacotes é igual a quantidade de caracteres na sequência de caracteres definida;
5. Permanece em valor alto por 0.5 s, reinicia a contagem de pacotes enviados e volta para o passo 3.

3.2.3 Rotina do receptor

Toda a temporização é realizada por interrupções por comparação de tempo, mantendo o mesmo intervalo de tempo (ciclo de *clock*) para cada *bit* recebido, exceto para o *bit* de início. A detecção do *bit* de início é realizada via interrupção externa por borda de descida. Detectado um *bit* de início, é aguardado 1.5 ciclos de *clock* para fazer a amostragem do dado recebido no centro da transmissão do *bit*, como no exemplo da [Figura 12](#).

Figura 12 – Exemplo de amostragem feita pelo receptor



Fonte: Autor.

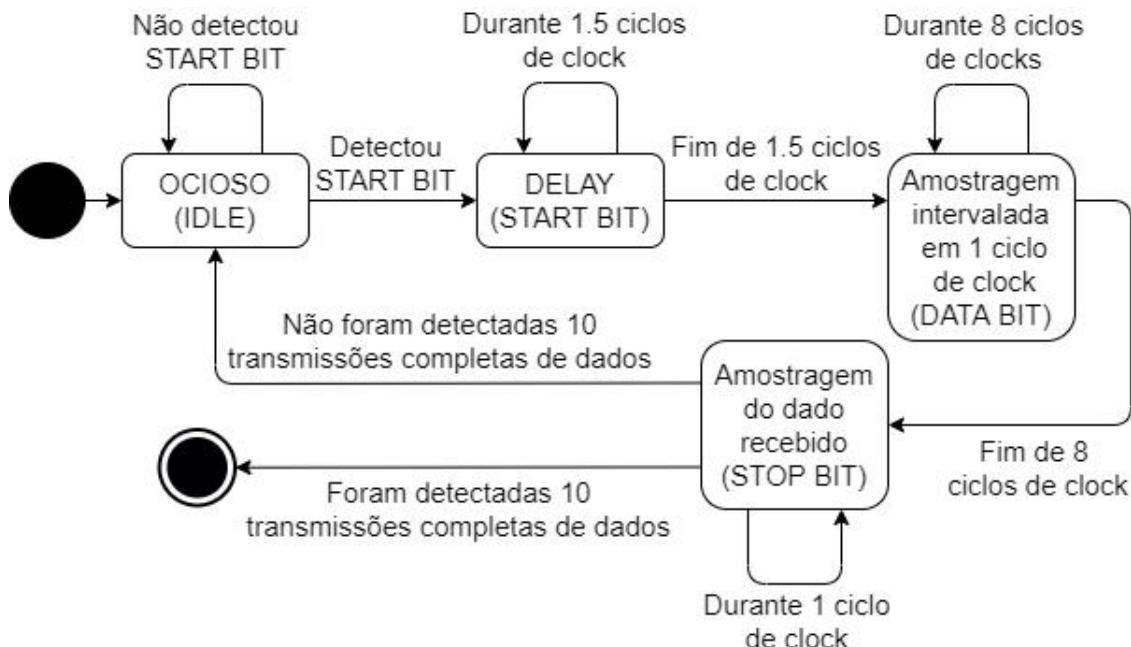
O receptor segue a seguinte ordem de execução:

1. Inicializa o sistema;
2. Aguarda a detecção de um *bit* de início;
3. Aguarda 1.5 ciclo de *clock*;
4. Faz a leitura de 9 valores recebidos (*bits*), cada uma intervalada em 1 ciclo de *clock*:
 - a) Os 8 primeiros valores correspondem ao valor do dado (*byte*) enviado pelo transmissor;
 - b) O último *bit* corresponde ao *bit* de parada;
5. Caso o *bit* de parada recebido seja de valor alto e o *byte* recebido seja alfanumérico, armazena o *byte* recebido em um *buffer*. Caso contrário armazena ‘?’ no *buffer*;
6. Caso a quantidade de bytes recebidos seja igual à quantidade de caracteres presentes na *string* utilizada para transmissão, libera o *buffer* no monitor Serial do Arduino e soma 1 na quantidade de *strings* recebidas;

7. Volta para o passo 2 se a quantidade de *strings* recebidas (transmissões completas) for menor ou igual a 10;
8. Finaliza o programa em execução ao detectar 10 transmissões completas.

Um diagrama de máquina de estado para o módulo receptor é apresentado na [Figura 13](#). No programa, todos os tempos são calculados por meio de interrupções por comparação do *timer*.

Figura 13 – Máquina de estados para o módulo receptor



Fonte: Autor.

3.3 DESCRIÇÃO DETALHADA DO TRANSMISSOR

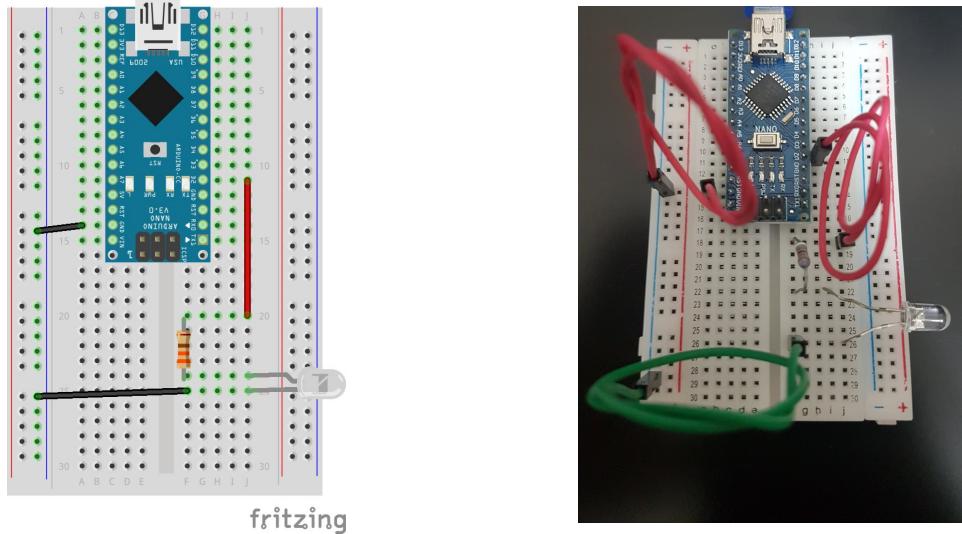
Nessa seção, é apresentada a implementação do *hardware* do transmissor e o algoritmo utilizado por esse.

3.3.1 Hardware do transmissor

O transmissor apresentou uma única configuração de *hardware* ao longo de todo o projeto. A lista de materiais utilizados está no [Seção A.2](#)

Basicamente, o pino 2 (D2) do Arduino foi conectado ao resistor de $330\ \Omega$, o qual foi devidamente conectado ao LED, sendo esse último aterrado de forma apropriada. O pino 2 corresponde à Porta D, no *bit* 2, de entrada e saída digital, a qual foi configurada como saída. Um esquemático e a versão física são mostrados na [Figura 14](#).

Figura 14 – Esquemático e implementação física do *hardware* transmissor



Fonte: Autor.

3.3.2 Algoritmo do transmissor

As *strings* utilizadas como dados ao longo das transmissões, as quais foram inseridas manualmente no código do transmissor, estão listadas no [Apêndice C](#). As sequências de caracteres foram colocadas manualmente, pois o Arduino não tem acesso direto aos arquivos do sistema operacional.

Como observado ao longo da descrição do *hardware*, o módulo emissor e o módulo receptor não compartilham de um meio de sincronização de dados, ficando a cargo do programador inserir os valores corretos de frequência e tamanho da *string* diretamente no algoritmo desenvolvido, para ambos os programas, a fim de manter a sincronia entre os módulos.

Os códigos inseridos nos [Apêndice D](#) foram levemente alterados com relação a sua estrutura quando comparado com o código fonte para melhor visualização na leitura desse texto. Porém, nenhuma parte de execução foi removida. As alterações foram apenas pequenos agrupamentos de linhas e remoção de alguns comentários, já que os comentários basicamente explicavam o que é mencionado ao longo desse texto. Os códigos fonte podem ser encontrados em <https://github.com/danncosta97/arduino-vlc>.

3.3.2.1 Considerações iniciais, macros e variáveis globais

Nos comentários iniciais do [Código D.1](#), as partes mais importantes são com relação ao mapeamento do pino 2 do Arduino Nano, o qual é acessado via Porta D

pelo *bit* 2. Além disso, uma ressalva deve ser feita com relação à versão 1.18.6 da IDE do Arduino utilizada, já que em versões mais novas ocorreram erros ao carregar o programa para a placa.

Após os comentários iniciais nesse código, temos a definição de algumas macros. Essas macros foram definidas principalmente para facilitar a manipulação e compreensão do código. A primeira sequência de macros é utilizada como base para executar a máquina de estados (*START_STATE* 0, *START_BIT_STATE* 1, *DATA_BITS_STATE* 2 e *STOP_BIT_STATE* 3), como na [Figura 11](#).

A macro *STRING_SIZE* é utilizada como base para alocar memória para um vetor de caracteres, onde é armazenado a *string* a ser transmitida.

As últimas duas macros são utilizadas para controle da frequência de transmissão. A macro *FREQUENCY* nada mais é do que a frequência desejada em *Hertz*. Já *OCR1A_FROM_FREQUENCY* é o valor a ser carregado no registrador OCR1A. Relembrando, OCR1A é o registrador de referência a ser comparado com o valor do *TIMER/COUNTER* 1 no Canal A (ver [Seção 2.7.1.1.3](#)). Sabendo que o *clock* utilizado pelo Arduino é de 16 MHz, temos que o período ou ciclo do *clock* (T_{clock}) é;

$$T_{clock} = \frac{1}{16 \text{ MHz}} = 62.5 \text{ ns}. \quad (2)$$

Se o valor da frequência desejada for de $f_{desejada}$ [Hz], precisamos então de um período de

$$T_{clockDesejado} = \frac{1}{f_{desejada}} = x \cdot 62.5 \text{ ns}, \quad (3)$$

onde x corresponde a quantidade de vezes que 1 período de *clock* do sistema deve ser considerado, a fim de obter a frequência desejada do *clock* para a implementação.

Assim, temos que

$$T_{clockDesejado} = T_{clock} \cdot x \Rightarrow \frac{1}{f_{desejada}} = \frac{1}{16 \text{ MHz}} \cdot x \Rightarrow x = \frac{16 \text{ MHz}}{f_{desejada}}, \quad (4)$$

tal que x corresponde à quantidade de vezes que 1 período de *clock* do sistema, um *clock* com frequência de 16 MHz no caso, deve ser contado para obter-se o valor do período do *clock* desejado.

Como OCR1A (ver [Seção 2.7.1.1.3](#)) é o registrador de referência utilizado na comparação, ajustamos a frequência da implementação atribuindo à OCR1A o valor de

$$\text{OCR1A} = \frac{16 \text{ MHz}}{f_{desejada}}, \quad (5)$$

o que permite criar uma interrupção por comparação com o *TIMER/COUNTER* 1 no canal A, já que uma contagem ascendente é feita no registrador TCNT1. Foi utilizado

o *clock* direto do Arduino pelo fato ser utilizado um *prescaler* de 1. Como o valor atribuído pode ser um número real, o próprio compilador do algoritmo se encarrega de considerar apenas o valor inteiro do número.

Pelos motivos listados acima, a macro *OCR1A_FROM_FREQUENCY* não deve ser alterada, caso contrário, a frequência não corresponderá à desejada.

Em seguida, temos algumas variáveis globais:

- *string*: vetor de caracteres onde é armazenada a sequência de caracteres a ser transmitida;
- *stringLengthSent*: armazena a quantidade de caracteres enviados de uma *string*;
- *state*: armazena o estado atual do sistema com base nas macros de estado do sistema;
- *stateChangeLocker*: variável booleana que impede a execução do código dos estados:
 - É alterada para verdadeiro frente a uma interrupção;
 - É alterada para falso frente a uma execução de código de um estado;
- *dataByteIndex*: armazena o índice do *bit* enviado no vetor de *bits* do *byte* (caractere) atualmente em transmissão;
- *systemStartCounter*: utilizado para fazer contagens genéricas, mais especificamente auxilia na contagem do tempo de inicialização do sistema (5 s) e intervalos entre transmissões (0.5 s);
- *finishedTransmissions*: armazena a quantidade strings enviadas (transmissões feitas).

Com relação à variável *stateChangeLocker*, ela apresenta o comportamento descrito acima devido ao fato de que um estado deve ser executado uma única vez por ciclo. Como o *clock* da implementação é gerenciado via interrupções, sempre que uma interrupção ocorre significa que executaremos um estado da máquina de estados. Como o Arduino executa sua rotina de *loop* a todo instante, uma forma de não permitir que os estados sejam executados sem o controle temporal desejado é de fato inserindo uma variável que os bloquee. Assim, sempre que um estado é executado, esse estado bloqueia, por meio da alteração da variável de bloqueio dentro do código do estado, a execução dos outros estados. E essa execução só é liberada novamente em uma interrupção.

3.3.2.2 Inicialização de variáveis e configuração de registradores

A partir do [Código D.2](#), há as definições iniciais do programa e configurações dos registradores. Inicialmente a comunicação com o monitor Serial é habilitada e o LED embutido na placa Arduino é desligado.

Em seguida, o pino 2 é definido como saída com valor alto. As variáveis globais anteriormente detalhadas são devidamente inicializadas. O estado inicial é colocado como *START* e é liberada a execução dos estados.

Ao longo do [Código D.2](#), as interrupções globais são desativadas e os registradores são configurados de modo que seja habilitada a interrupção por comparação do *TIMER/COUNTER* 1 com base em OCR1A no canal A e o valor atual do mesmo contador (TCNT1) é zerado. Além disso, o *prescaler* é definido como 1, ou seja, (sem *prescaler*). Ao final as interrupções globais são reativadas.

3.3.2.3 Rotina *loop()*

Dentro da rotina de *loop()*, mostrada no [Código D.3](#), contanto que *stateChangeLocker* seja verdadeiro (1), há a execução dos códigos dos estados, com base no estado atual. As funções *Serial.Print()* são apenas para *debug*. Algo comum a todos estados é a alteração de *stateChangeLocker* para falso (0) ao fim de sua execução, como explicado anteriormente na [Seção 3.3.2.1](#).

No caso do *START_STATE*, é assegurado que o valor de saída do pino 2 seja alto.

Já no *START_BIT_STATE*, há a escrita do valor baixo em todas as portas D. Esse procedimento é realizado pois todos os outros *bits* da porta não estão sendo utilizados e, consequentemente, não causam alterações no fluxo de execução.

No estado *DATA_BITS_STATE*, há a escrita do valor dos *bits*, *bit* a *bit*, do *byte* (caractere) atualmente sendo enviado. Como um *bit* é enviado por ciclo, *dataByteIndex* inicialmente armazena o valor atual do *bit* para envio e, no final, é acrescido de 1 para indicar o índice do próximo *bit* a ser enviado. O envio dos *bits* é feito do menos significativo para o mais significativo. O valor de saída do pino 2 é alterado de acordo com o valor do *bit* atual. Caso o valor de *dataByteIndex* seja maior ou igual a 8 ao final da execução desse estado, significa que um *byte* completo acaba de ser enviado, e assim, a variável que guarda a quantidade de caracteres enviados (*stringLengthSent*) é acrescido de 1.

O último estado, *STOP_BIT_STATE*, apenas realiza a alteração da saída do pino 2 para valor alto, indicando um *bit* de parada.

3.3.2.4 Rotina de serviço de interrupção

A última parte do código ([Código D.4](#)), é a rotina de interrupção para interrupção por comparação do *TIMER/COUNTER* 1 no Canal A, como indicado no parâmetro da função *ISR()*.

Inicialmente a rotina verifica se o estado atual é de inicialização do sistema (*START_STATE*). Se for, decresce o valor de *systemStartCounter*. Como *systemStartCounter* possui o valor da frequência desejada para o módulo multiplicada por “n” segundos e a interrupção ocorre a cada período da frequência desejada, o estado ficará travado em (*START_STATE*) por “n” segundos. Uma vez que *systemStartCounter* chega a 0, é alterado o estado atual do sistema para *START_BIT_STATE*.

Caso o estado no início da ISR seja *START_BIT_STATE*, apenas é feita a atualização do estado para *DATA_BITS_STATE*.

Já para um início da ISR com estado de *DATA_BITS_STATE*, é feita a verificação do índice do próximo *bit* a ser enviado do atual *byte* em transmissão. Caso o índice seja maior ou igual a 8, o atual *byte* foi totalmente enviado e *dataByteIndex* é zerado para o próximo *byte* e é feita a troca para o estado *STOP_BIT_STATE*. Caso o *byte* não tenha sido totalmente enviado, apenas é liberada a execução do código dos estados novamente para o estado de envio de dados (*DATA_BITS_STATE*).

Em último caso, sendo o estado atual o do *bit* de parada (*STOP_BIT_STATE*), se a quantidade de caracteres enviados for igual ao tamanho da *string* de envio, retornaremos o módulo para o estado de inicialização (*START_STATE*), porém, *systemStartCounter* com o fator de multiplicação da frequência de 0.5, ou seja, agora o sistema ficará em valor alto por 0.5 s. Dessa forma o sistema fica sempre com um intervalo de 0.5 s entre transmissões. Caso a quantidade de caracteres enviados não seja igual ao tamanho da *string* de envio, o estado é alterado para o de *bit* de início, (*START_BIT_STATE*), já que ainda há caracteres a serem enviados.

3.4 DESCRIÇÃO DETALHADA DO RECEPTOR

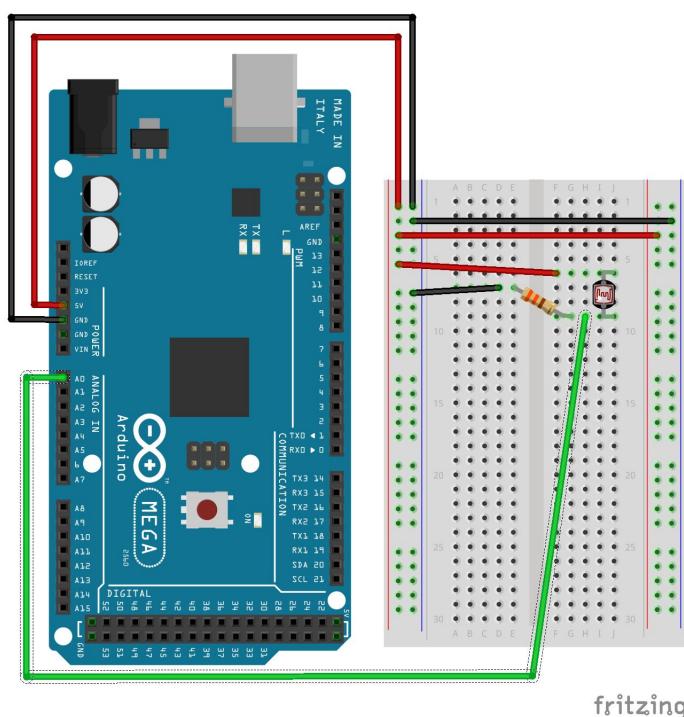
Ao longo dessa seção será descrita as implementações realizadas para o receptor até atingir a configuração final. Além disso, é descrito o algoritmo implementado para o receptor.

3.4.1 Hardware Receptor

O receptor teve sua disposição de componentes alterada algumas vezes até atingir o modelo final. Da mesma forma que para o *hardware* do transmissor, o funcionamento correto dos componentes foi verificado através de um multímetro digital.

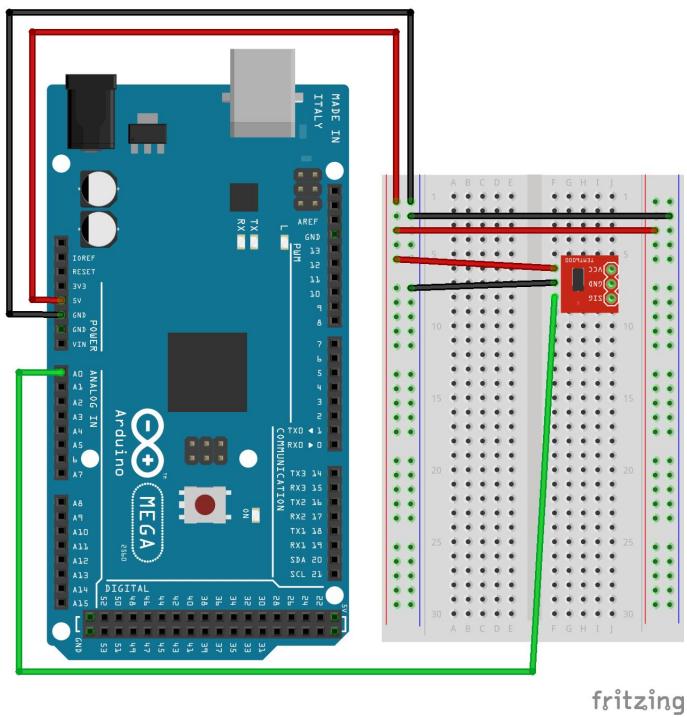
O receptor foi inicialmente implementado com a utilização de um LDR com saída conectada ao pino A0 de um Arduino Mega Rev3, como na [Figura 15](#). Apesar da comunicação ter sido estabelecida com a utilização do LDR, por decisão de projeto, o LDR foi substituído pelo módulo TEMT6000, contendo um fototransistor sensível à luz ambiente, como na [Figura 16](#).

Figura 15 – Esquemático da implementação inicial do hardware receptor com LDR



Fonte: Autor.

Figura 16 – Esquemático da implementação inicial do *hardware* receptor com Módulo TEMT6000



Fonte: Autor.

Essas versões foram abandonadas por utilizarem a porta analógica como entrada e teoricamente utilizarem mais tempo de processamento para processar o valor de um sinal. A vantagem dessas implementações seria uma simplificação do circuito, deixando para o algoritmo determinar quando houvesse o recebimento de um *bit* de valor ‘0’ ou de valor ‘1’.

Na implementação final com o módulo TEMT6000, mostrada na Figura 16, a saída do módulo passou agora a ser conectado à entrada não inversora de um amplificador operacional LM358N, o qual por sua vez foi alimentado com 5 V na sua alimentação positiva e 0 V na sua alimentação negativa. À entrada inversora do amplificador utilizado, foi conectada a saída de um divisor de tensão, sendo a tensão de referência para o amplificador utilizado em modo comparador. Com isso, a saída do amplificador foi conectada ao pino 2 do Arduino (Porta E *bit* 4), um pino digital, o qual foi configurado como entrada. A escolha pelo amplificador operacional LM358 ocorreu por ser uma alternativa melhor ao UA741, onde o LM358 apresentou uma menor diferença entre a tensão de saída e a tensão de alimentação. A lista de componentes utilizados no receptor final está no Seção A.3.

Como o amplificador operacional foi utilizado em seu modo comparador, o recebimento de *bit* com valor alto ou valor baixo passou a ser definido via *hardware*, em

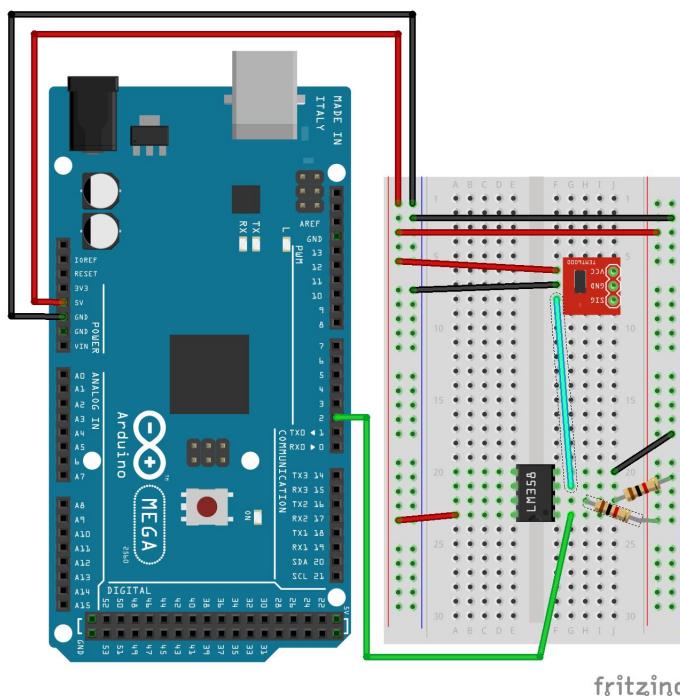
que o pino 2 recebe, efetivamente, apenas os valores de 5 V ou 0 V.

Como o fototransistor é sensível a intensidade de luz incidida, quanto maior a distância entre os módulos transmissor e receptor, menor a intensidade de luz recebida pelo fototransistor e menor a tensão de saída do módulo TEMT6000. Assim, uma forma de configurar a distância máxima de transmissão é através da tensão de referência no comparador. Uma maior tensão de referência no comparador gera uma menor distância máxima de transmissão entre os módulos.

Por isso, a coleta de dados foi realizada com três configurações de divisor de tensão diferentes para uma entrada de 5 V. Assim, na entrada não inversora do amplificador operacional foram conectadas as configurações de divisor de tensão $R_1 = 1\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$, $R_1 = 10\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$ ou $R_1 = 50\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$ de acordo com a coleta de dados. Os 50 k Ω foram alcançados através da utilização dos dois resistores de 100 k Ω conectados em paralelo.

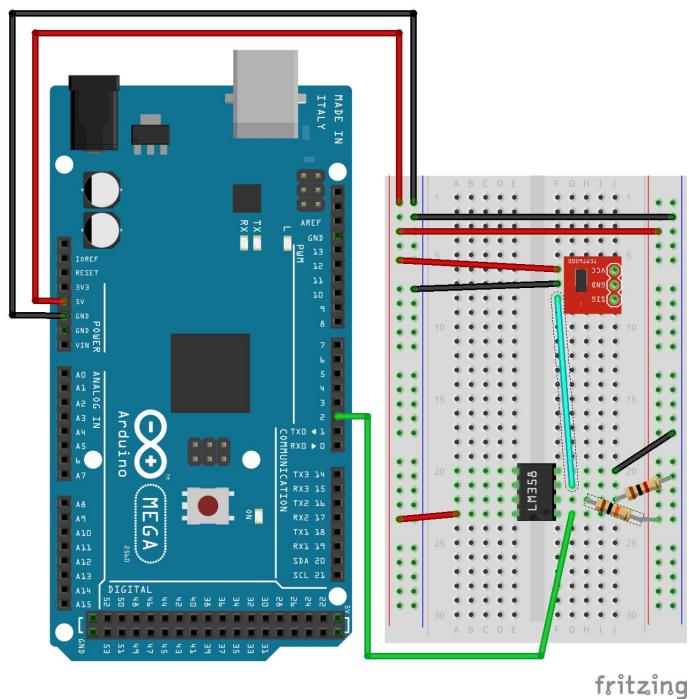
A [Figura 17](#), a [Figura 18](#) e a [Figura 19](#) mostram um esquemático das três possíveis configurações finais para o módulo receptor. Já a [Figura 20](#) mostra a implementação física da configuração com uso de $R_1 = 50\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$. As demais configurações reais seguem o mesmo modelo, alterando apenas o divisor de tensão.

Figura 17 – Esquemático da implementação final do *hardware* receptor com divisor de tensão feito com $R_1 = 1\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$



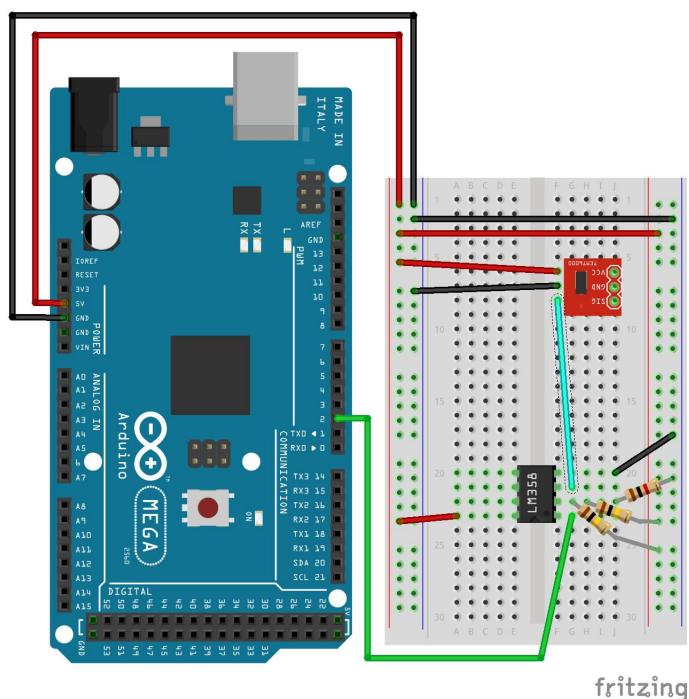
Fonte: Autor.

Figura 18 – Esquemático da implementação final do *hardware* receptor com divisor de tensão feito com $R_1 = 10\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$



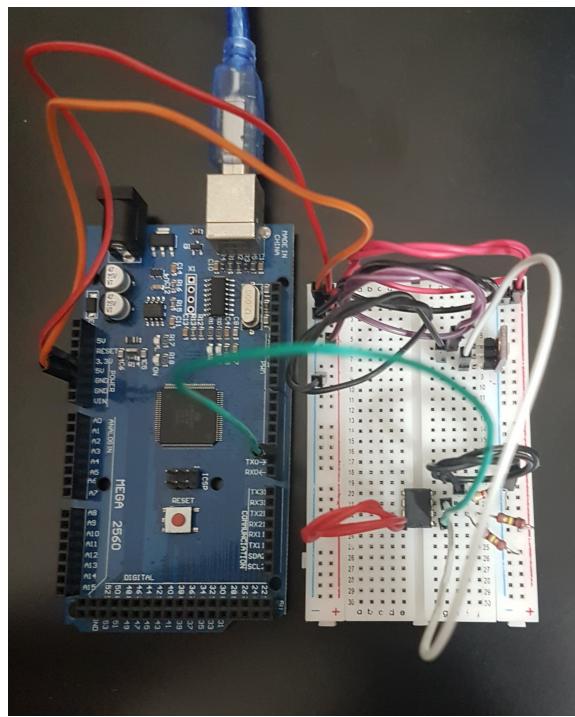
Fonte: Autor.

Figura 19 – Esquemático da implementação final do *hardware* receptor com divisor de tensão feito com $R_1 = 50\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$



Fonte: Autor.

Figura 20 – Implementação final física do *hardware receptor* com divisor de tensão feito com $R_1 = 50\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$



Fonte: Autor.

3.4.2 Algoritmo do receptor

Nessa seção é descrito o algoritmo implementado para o receptor, a fim de executar a máquina de estados presente na [Figura 13](#). As mesmas ressalvas feitas no início da [Seção 3.3.2](#) aplicam-se também a esta seção. O código descrito aqui encontra-se no [Apêndice E](#).

3.4.2.1 Considerações iniciais, macros e variáveis globais

Nos comentários iniciais do [Código E.1](#), as partes mais importantes são com relação ao mapeamento do pino 2 do Arduino Mega Rev3, o qual é acessado via Porta E pelo *bit 4* (ver [Seção 2.7.1.1.3](#)). Além disso, a mesma ressalva é feita com relação à versão 1.18.6 da IDE do Arduino utilizada, já que em versões mais novas ocorreram erros ao carregar o programa para a placa.

Assim como no código do transmissor, temos a definição de algumas macros. A primeira sequência de macros será utilizada como base para executar a máquina de estados (*IDLE_STATE* 0, *START_BIT_STATE* 1, *DATA_BITS_STATE* 2 e *STOP_BIT_STATE* 3), como na [Figura 13](#).

As macros *STRING_SIZE*, *FREQUENCY* e *OCR1A_FROM_FREQUENCY* seguem o mesmo padrão descrito em [Seção 3.3.2.1](#).

Apenas para recordar, a detecção do *bit* de início será feita via interrupção por borda de descida. Como é necessário amostrar no meio do *bit*, é preciso inserir inicialmente um *delay* de meio ciclo após a detecção do *bit* de início. Posteriormente, deve-se considerar os ciclos completos. Com isso, a macro *OCR1A_FROM_FREQUENCY_HALF* contém o valor de metade das contagens necessárias dos ciclos do *clocks* do Arduino de 16 MHz para que as interrupções por tempo gerem a frequência desejada na macro *FREQUENCY*.

Dessa forma, frente a uma borda de descida de um *bit* de início, a máquina será alterada para *START BIT* com o contador de *COUNTER/TIMER 1*, já iniciado com o valor de *OCR1A_FROM_FREQUENCY_HALF*. Assim realizará meio ciclo do *clock* desejado sem trocar de estado e os próximos períodos serão períodos completos, com base na frequência desejada, e agora sim os estados serão alterados ao fim de um ciclo. Com isso, a amostragem dos dados efetivos e do *bit* de parada é devidamente defasada em meio ciclo, amostrando o centro do sinal de transmissão do *bit*.

Seguindo para a última macro, temos *TRANSMISSION_LIMIT*, que nada mais é do que o parâmetro utilizado para encerrar o programa em execução, contendo o limite máximo de transmissões para receber (quantidade máxima de strings para receber).

Em seguida, temos algumas variáveis globais

- *initialdelay*: variável booleana que controla a execução do estado de *bit* de início por meio ciclo a mais, para defasar os ciclos em meio período;
- *stringReceivedBuffer*: vetor de caracteres para armazenar os *bytes* (caracteres) recebidos. Tamanho máximo de 1300 *bytes* devido a memória do transmissor, o qual suportou armazenar pouco mais que 1300 *bytes* de dados para transmissão;
- *charBits*: *bits* recebidos de um *byte*, concatenados. O caractere de fato recebido da transmissão;
- *bitReceived*: valor do *bit* atual recebido;
- *bytesReceivedAmount*: quantidade de *bytes* recebidos. Utilizado como parâmetro para determinar o recebimento completo de uma sequência de caracteres.
- *state*: armazena o estado atual do sistema com base nas macros de estado do sistema;
- *stateChangeLocker*: variável booleana que impede a execução do código dos estados:

- É alterada para verdadeiro frente a uma interrupção;
- É alterada para falso frente a uma execução de código de um estado;
- *dataByteIndex*: armazena o índice do *bit* enviado no vetor de *bits* do *byte* (caractere) atualmente em transmissão;
- *finishedTransmissions*: armazena a quantidade strings enviadas (transmissões feitas).

3.4.2.2 Inicialização de variáveis e configuração de registradores

A partir do [Código E.2](#), há as definições iniciais do programa e configurações dos registradores. Partes idênticas ao [Código D.2](#) tiveram os comentários ao longo do código removidos para visualização em uma única página. Inicialmente a comunicação com o monitor Serial é habilitada e o LED embutido na placa Arduino é desligado.

Em seguida, o pino 2 é definido como entrada. As variáveis globais anteriormente detalhadas são devidamente inicializadas e no *buffer stringReceivedBuffer* é inserido ‘\0’ na última posição do vetor, indicando fim de *string*. O estado inicial é colocado como *IDLE* e é bloqueada a execução dos estados, já que o gatilho para execução dos estados será a borda de descida do *bit* de início.

Ao longo do [Código E.2](#), as interrupções globais são desativadas e os registradores são configurados de modo que seja habilitada a interrupção por comparação do *TIMER/COUNTER 1* com base em *OCR1A* no canal A e o valor atual do mesmo contador (*TCNT1*) é zerado. Além disso, o *prescaler* é definido como 1 (sem *prescaler*).

Diferentemente do transmissor, temos agora que configurar a interrupção externa. Como já mencionado na [Seção 2.7.1.1.2](#), os registradores *EICRx* são responsáveis por tal configuração. Com base no mapeamento de portas e pinos do Arduino Mega Rev3, observa-se que *EICRB[1:0]* configura o modo da interrupção externa em *INT4*, o qual corresponde ao pino 2 da placa. Com isso, o pino 2 é configurado para executar uma interrupção frente a uma borda de descida. Em seguida, via *EIMSK[4]*, as interrupções externas são ativadas ao pino correspondente à *INT4*, pino 2. Ao final as interrupções globais são reativadas.

3.4.2.3 Rotina *loop()*

Dentro da rotina de *loop()*, contanto que *stateChangeLocker* seja verdadeiro (1), há a execução dos códigos dos estados, com base no estado atual. A função *Serial.Print("FINISHED")* serve apenas para visualização do recebimento limite de transmissões. Algo comum a todos estados é alteração de *stateChangeLocker* para

falso (0) ao fim de sua execução, como explicado anteriormente na [Seção 3.3.2.1](#). O código [Código E.3](#) teve algumas linhas agrupadas, sem alteração do fluxo de execução ou remoção de partes, para que o código possa ser mostrado em uma única página.

No caso do *IDLE*, é apenas um estado vazio, de fato ocioso, o qual aguarda uma interrupção por borda de descida.

No *START_BIT_STATE*, há a desativação das interrupções externas, já que agora as interrupções devem ser feitas através da comparação do *TIMER/COUNTER 1*, para que a frequência desejada seja respeitada.

No estado *DATA_BITS_STATE*, é feito a leitura do pino 2 e seu valor é armazenado em *bitReceived*. De acordo com o índice do *bit* do *byte* recebido, o *bit* é devidamente armazenado em *charBits*. Por fim o índice do próximo *bit* a ser recebido é acrescido de 1.

O último estado, *STOP_BIT_STATE*, realiza uma validação do *bit* recebido como *bit* de parada. Caso seja um valor alto e o valor contido for um caractere alfanumérico com base na tabela ASCII, é armazenado o valor em *charBits* na posição correta do buffer *stringReceivedBuffer*, de acordo com *bytesReceivedAmount*. Caso *charBits* não contenha um alfanumérico, um “?” é inserido no *buffer*. Considera-se nesse caso que tivemos um falso positivo valor alto para *bit* de parada. Inicialmente, se o valor do *bit* de parada for um valor baixo, temos um erro de transmissão/recebimento de *byte* e “?” é inserido no *buffer*. Por fim, *bytesReceivedAmount* é acrescido de 1.

Seguindo no mesmo estado, caso *bytesReceivedAmount* seja maior do que o tamanho da *string* definida para transmissão, há a liberação do buffer *stringReceivedBuffer* no monitor Serial do Arduino. A quantidade de transmissões finalizadas, strings completamente recebidas, também é acrescida em 1. Continuando, *charBits* é zerado, preparando para o próximo *byte*.

Ainda com relação ao último estado, *STOP_BIT_STATE*, as interrupções externas são reativadas para INT4 via registrador EIMSK, já que o sistema deve aguardar o próximo *bit* de início. Por fim o estado é alterado para *IDLE*. Como as interrupções externas foram reativadas no meio do *bit* de parada, junto com a amostragem, o código do estado de *IDLE_STATE* só ocorrerá se:

- *bit* de parada teve valor alto e o próximo *bit* tem valor alto;
- *bit* de parada teve valor baixo e o próximo *bit* tem valor baixo;
- *bit* de parada teve valor baixo e o próximo *bit* tem valor alto.

Isso ocorre devido ao fato de que não há borda de descida nos casos listados e a interrupção por comparação do *timer* será lançada primeiro. Caso contrário, a

interrupção externa por borda de descida ocorrerá primeiro e o estado já será alterado para *START_BIT_STATE*.

Uma observação é que um erro de *bit* de parada pode ter ocorrido tanto por erro de leitura do *bit* de parada quanto do *bit* de início ou por um erro de envio de dado. A conclusão é que todos os casos ocasionarão em uma falha da transmissão de um *byte*, o que pode acarretar em indefinidos erros sucessivos.

3.4.2.4 Rotina de serviço de interrupção

A última parte do código ([Código E.4](#)) é a rotina de interrupção para interrupção externa no pino 2 e para interrupção por comparação do *TIMER/COUNTER* 1 no Canal A, como indicado em *ISR(INT4_vect)* e *ISR(TIMER1_COMPA_vect)*, respectivamente.

No caso da interrupção externa, que ocorre devido à borda de descida no pino 2, o estado da máquina é alterado para *START_BIT_STATE* e ao contador do *TIMER/COUNTER* 1 é atribuído o valor de *OCR1A_FROM_FREQUENCY_HALF*. Além disso, o valor de *initialdelay* é alterado para verdadeiro. Já *stateChangeLocker* é alterado para falso, para que não haja execução de código de nenhum estado. Assim o sistema apenas aguarda uma interrupção por comparação do *timer*, onde ocorrerá o atraso de meio ciclo de acordo com a variável *initialdelay*.

Seguindo para a interrupção por comparação do *timer*, inicialmente há a verificação do atual estado do sistema. Caso o sistema esteja ocioso, é liberado a execução do código de ociosidade, o qual não realiza nenhuma ação, a não ser bloquear a execução do código dos estados e aguardar uma interrupção por tempo.

Caso a interrupção do *timer* ocorra e estado atual seja do *bit* de início, *START_BIT_STATE*, a variável booleana *stateChangeLocker* é alterada para 1. Caso a variável *initialdelay* seja verdadeira (1), significa que a interrupção ocorreu logo após o *delay* inicial de meio ciclo de *clock*. Assim, o estado é mantido, mas a variável é alterada para falso (0). Caso contrário, o código do estado já foi executado, bloqueando interrupções externas e o sistema está no centro de um *bit* de dado efetivo. O estado é então alterado para o estado de leitura de dados efetivos, *DATA_BITS_STATE*.

Se a interrupção ocorrer e o estado atual for de leitura de dados, é feita a verificação de *dataByteIndex*. Caso *dataByteIndex* seja maior ou igual a 8 significa que todos os *bits* do *byte* foram lidos e o *bit* a ser amostrado é o de parada. Assim o estado é alterado para *STOP_BIT_STATE*. Sempre que esse cenário ocorre, também é liberado a execução do código dos estados.

A interrupção não analisa o caso do estado atual ser de um *bit* de parada,

devido a explicação dada para o caso do estado de *bit* de parada no [Código E.3](#), na [Seção 3.4.2.3](#).

3.5 COLETA, COMPILAÇÃO E ANÁLISE DE DADOS

3.5.1 Método de coleta de dados

Com o *hardware* e algoritmo finalizados, iniciou-se a coleta de dados. Primeiramente, foram definidos 3 sequências de caracteres alfanuméricos e de diferentes tamanhos, 100, 500 e 1000 caracteres, as quais foram utilizadas como dados durante a transmissão.

Observou-se que alguns parâmetros influenciavam na taxa de erros de uma transmissão. Os parâmetros foram:

- a tensão de referência do amplificador operacional utilizado no modo comparador;
 - a quantidade de caracteres transmitidos;
 - a distância entre as partes comunicantes (emissor e receptor);
 - a frequência de transmissão.

Assim, o método de coleta de dados seguiu o pseudocódigo de laços aninhados no [Código 3.1](#):

Código 3.1 – Rotina de obtenção de dados

```
1 Enquanto houver uma nova tensão de referência {
2     Enquanto houver uma nova sequência de caracteres alfanuméricos
3         {
4             Enquanto houver uma nova distância em que haja transmissão
5                 de dados {
6                 Enquanto houver uma nova frequência em que haja
7                     transmissão de dados {
8                         Coleta 10 transmissões intervaladas em 0.5s;
9                     }
10                }
11            }
12        }
13    }
```

Fonte: Autor.

Os parâmetros utilizados nos laços do Código 3.1 foram:

- **Tensão de referência:** a tensão de referência V_{ref} foi obtida através da tensão de saída de 3 configurações de divisor de tensão para um tensão de entrada de 5 V:
 - Configuração 1: $R_1 = 1 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$: $V_{ref} = 2.5 \text{ V}$;

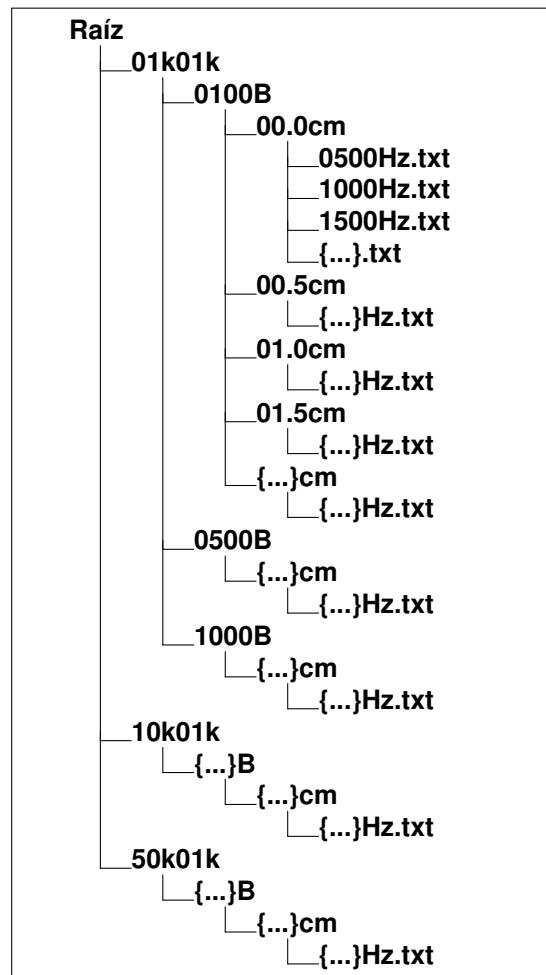
- Configuração 2: $R_1 = 10 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$: $V_{\text{ref}} \approx 0.46 \text{ V}$;
- Configuração 3: $R_1 = 50 \text{ k}\Omega$ e $R_2 = 1 \text{k}\Omega$: $V_{\text{ref}} \approx 0.10 \text{ V}$.

- **Sequência de caracteres alfanuméricos:**

- 100 caracteres (100 *Bytes*), 500 caracteres (500 *Bytes*) ou 1000 caracteres (1000 *Bytes*). As cargas de dados de 100, 500 e 1000 *bytes* foram definidas não só pelo limite de memória no Arduino Nano, mas também para manter uma proporção de 1:5:10.
- **Distância:** a distância inicial foi de 00.0 cm e foi incrementada em 00.5 cm sempre que necessário.
- **Frequência:** a frequência inicial foi de 500 Hz e foi incrementada em 500 Hz sempre que necessário.

Todas as 10 transmissões realizadas eram salvas em arquivos de texto contendo o valor da frequência como nome (xxxxHz.txt) dentro de diretórios com o nome contendo o valor da distância (xx.xxcm), os quais por sua vez estavam dentro de diretórios com o nome contendo o valor de *bytes* transmitidos (xxxxB), os quais por fim estavam dentro de diretórios com o nome contendo a configuração do divisor de tensão (xxkxxk). Um esquemático da árvore de diretórios e arquivos é mostrado na [Figura 21](#).

Figura 21 – Esquemático da árvore de diretórios e arquivos gerada com os dados coletados



Fonte: Autor.

3.5.2 Método de análise dos dados

O método de análise dos dados foi feito de acordo com a quantidade de transmissões aceitas como válidas. Transmissões que façam o recebimento de 100% dos dados enviados, ou seja, apresentem 0% de erro de transmissão, foram consideradas transmissões “limpas”. Aquelas com uma taxa de erro maior que 0% e menor ou igual a 5% foram consideradas transmissões “sujas”. Transmissões com demais taxas de erro foram consideradas como “falhas”. Um resumo é mostrado no [Quadro 5](#).

O conjunto de 10 transmissões foi considerado “bem sucedido”, “aceitável” ou “falho” para os parâmetros analisados de acordo com o [Quadro 6](#)

A partir disso, foram tabelados os resultados e feitas as devidas avaliações acerca das variáveis estipuladas para cada conjunto de dados.

Quadro 5 – Nomenclatura de uma transmissão com base na taxa de erro a nível de bytes

Nomenclatura de uma transmissão com base na taxa de erro	Taxa de erro de transmissão (nível de byte)
Limpa	00.00 %
Suja	> 00.00 % e <= 05.00 %
Falha	> 05.00%

Fonte: Autor.

Quadro 6 – Consideração acerca do conjunto de 10 transmissões

Quantidade de tipo de transmissões não falhas	Resultado do conjunto de transmissões
10 transmissões limpas	Bem sucedido
Transmissões limpas < 10 transmissões falhas < 2	Aceitável
Transmissões falhas ≥ 2	Falho

Fonte: Autor.

3.5.3 Compilação dos dados

Para a compilação dos resultados obtidos, foi implementado um algoritmo em *Python 3.10*. O algoritmo é responsável por realizar uma grande base de dados com as transmissões realizadas, com diversas informações. A motivação para elaborar o algoritmo foi devido a quantidade de arquivos com transmissões gerados, efetivamente mais de 1800, e portanto seria praticamente inviável fazer uma compilação dos dados manualmente para análise.

O algoritmo consiste em coletar todos os arquivos *.txt* criados, analisar cada uma das transmissões do arquivo e retornar um *.json*, contendo informações da configuração do comparador, a quantidade de bytes, a distância, a frequência, índices de erros, a quantidade de transmissões limpas, sujas e falhas, bem como a conclusão acerca do conjunto de transmissões. As explicações aqui descritas serão feitas com base no [Código F.1](#), presente no [Apêndice F](#). Demais códigos citados ao longo desta seção estão também presentes em [Apêndice F](#).

O percorrimento na árvore de diretórios/arquivos é realizado com auxílio do módulo *OS* para Python via *import os*. Todos os arquivos abertos ao longo dos algoritmos são devidamente fechados.

As primeiras funções não geram/compilam dados algum, apenas fazem ajustes em nomes de diretórios e arquivos para que, durante a análise dos dados compilados,

haja uma ordenação alfabética. Elas não precisam ser de fato executadas, ficando a critério do usuário utilizá-las ou não. São elas:

- *renameToMillimeters()*: renomeia diretórios com nome em centímetros para milímetros. Exemplo: “5.0cm” é renomeado para “50mm”;
- *renameToCentimeters()*: renomeia diretórios com nome em milímetros para centímetros. Exemplo: “50mm” é renomeado para “5.0cm”;
- *adjustBytesName()*: renomeia diretórios com o nome correspondente ao número de bytes para apresentarem a quantidade com 4 dígitos. Exemplo: “100B” é renomeado para “0100B”; “1000B” não é alterado;
- *adjustCentimetersName()*: renomeia diretórios com o nome correspondente à distância em centímetros para apresentarem a distância com 2 dígitos inteiros e 1 dígito decimais. Exemplo: “5.0cm” é renomeado para “05.0cm”; “10.0cm” não é alterado;
- *adjustFrequencyName()*: renomeia arquivos com o nome correspondente à frequência em Hz para apresentarem a frequência com 4 dígitos. Exemplo: “500Hz” é renomeado para “0500Hz”; “1000Hz” não é alterado.

As funções *generateJsons()* e *compileJsons()* devem ser executadas em ordem, ao menos uma vez, para que a função *openJson()* seja executada corretamente. Executar *generateJsons()* e *compileJsons()* mais de uma vez apenas consumirá tempo de CPU.

3.5.4 Função *generateJsons()*

A função *generateJsons()* é a responsável por realizar a geração dos arquivos *json* para cada diretório de distância.

A função faz uma iteração sobre todos os arquivos presentes na árvore de diretórios com base em uma raiz, sendo o caminho local da raiz “C:/Users/Daniel/Documents/VLC_DADOS_CLEAN”. Como apenas os diretórios de distância possuem arquivos, cada um contendo ao menos 10 transmissões para uma determinada quantidade de bytes e para uma determinada configuração de divisor de tensão no comparador, esses serão os arquivos iterados.

A primeira condicional do laço é apenas para assegurar que apenas esses arquivos sejam levados em conta em uma segunda execução do programa, por exemplo.

No primeiro arquivo, é feita a abertura dele via *open*, e as suas linhas são colocadas em uma lista. Em seguida, variáveis contendo a frequência, a distância, a quantidade de *bytes* e a configuração do divisor de tensão do comparador são

inicializadas com os devidos valores. As variáveis que armazenarão os valores de transmissões limpas, sujas e falhas são também inicializadas.

A próxima condicional assegura que o arquivo lido tenha ao menos 10 transmissões gravadas.

Em um dicionário, são inseridos pares “chave:valor” (“*key:value*”) para a frequência, a distância, a quantidade de *bytes* e a configuração do divisor de tensão do comparador.

A seguir, há uma iteração sobre as 10 transmissões salvas na lista *transmissionsLines*. Para cada caso, é analisado a quantidade de *bytes* daquela transmissão através do nome do diretório pai do atual diretório, em que o pai contém a quantidade de *bytes* como seu nome. Sabendo a quantidade de *bytes*, o algoritmo pode então fazer a devida comparação da transmissão com a *string* originalmente utilizada para a transmissão.

Ao comparar a transmissão recebida (*string* recebida pelo receptor) e a *string* enviada pelo transmissor, são retirados os índices de erro da transmissão e de acordo com [Seção 3.5.2](#) é feita uma verificação para acrescer o valor de transmissões limpas, sujas ou falhas, a fim de determinar se o conjunto foi bem sucedido, aceitável ou falho. Todos esses dados são armazenados no dicionário com seus devidos pares de “chave:valor”. Ao final, o dicionário é convertido em um *json* e inserido em uma lista local ao laço “for” mais externo. Todos esse processo é repetido para cada arquivo presente no diretório de distância.

Uma vez que todos os arquivos de um diretório geraram seu *json* e o adicionaram na lista de *json* *transmissionsLineJsonArray*, essa lista de *json* é ajustada para criar um único arquivo “result.json” no diretório de distância. Assim, temos um compilado dos dados de cada frequência de transmissão para uma dada distância, a qual está ligada a uma quantidade de *bytes* transmitidos e que, por sua vez, está ligado a uma configuração de divisor de tensão, que gera uma tensão de referência, ligado ao amplificador operacional em modo comparador.

Como as interações são feitas sobre os arquivos, ao finalizar um diretório de distância, outro diretório de distância é acessado e um novo arquivo “result.json” é criado.

Caso arquivos “result.json” sejam encontrados em diretórios que não sejam de distância, eles são deletados.

Ao final da função, a quantidade de arquivos lidos é impressa para comparação manual via detalhes no sistema operacional.

3.5.5 Função `compileJsons()`

Com todos os arquivos “`result.json`” gerados em cada diretório de distância, esses arquivos são abertos em `compileJsons()`, iterando sobre arquivos com o módulo “OS”, para que um único arquivo “`result.json`” seja gerado no diretório raiz.

Cada um dos arquivos são abertos e convertidos em um dicionário, de tal forma que cada conjunto de dados para cada frequência em uma distância é lido e convertido para `json` e adicionado a uma lista. Uma vez que todos os arquivos “`result.json`” foram acessados, a lista contendo todos os `json` é gravada, fazendo alguns ajustes para ficar no formato `json`, em um único arquivo ‘`result.json`’ no diretório raiz.

Dessa forma um único arquivo apresenta os dados de todas as transmissões realizadas.

3.5.6 Função `openJson()`

A função `openJson()` abre o arquivo “`result.json`” gerado no diretório raiz.

Inicialmente os conjuntos de dados do arquivo são inseridos em um dicionário e é impresso na saída a quantidade de conjunto de dados presentes, o qual deve ser igual a quantidade de arquivos de frequência “`.txt`” lidos e presentes no sistema operacional.

Com os dados em um dicionário, iterando sobre e com uma condicional, é possível recuperar qualquer dado na saída. Por exemplo, o [Código F.2](#), retorna a frequência (“frequency”), a quantidade transmissões bem sucedidas das 10 realizadas (“overallTransmissions”) e o resultado das 10 transmissões (“resultTransmission”) da configuração, ajustada dentro da condicional, com o divisor de tensão do comparador formado por $R_1 = 1 \text{ k}\Omega + R_2 = 1 \text{ k}\Omega$, com uma carga de 100 *bytes* e uma distância de 00.0 cm.

Os parâmetros de busca da configuração são passados via condicional `if`, enquanto os parâmetros de saída são adicionados na variável `outputResultSearch` seguindo o modelo do código.

Para a busca feita no [Código F.2](#), o resultado obtido é mostrado em [Figura 22](#).

Como é possível observar, essa função é altamente customizável e deve ser modificada para realizar a busca dos dados desejados. Por exemplo, o [Código F.3](#), é utilizado para geração de uma parte dos dados do resultado apresentado na [Figura 24](#), presente na [Seção 4.1.1](#). Os dados gerados são apresentados na [Figura 23](#).

Na [Figura 23](#), há como saída a quantidade de conjunto de dados, a configuração atual buscada e os dados resultantes da busca. Para os dados resultantes, da

Figura 22 – Resultado gerado pelo Código F.2

```
1873
0500Hz 10/10 SUCCESS
1000Hz 10/10 SUCCESS
1500Hz 10/10 SUCCESS
2000Hz 10/10 SUCCESS
2500Hz 10/10 SUCCESS
3000Hz 10/10 SUCCESS
3500Hz 10/10 SUCCESS
4000Hz 00/10 FAILED
```

Fonte: Autor.

esquerda para a direita, a primeira coluna apresenta a frequência. As três colunas seguintes são a quantidade de transmissões limpas, sujas e falhas, respectivamente. A penúltima coluna apresenta a soma de transmissões limpas e sujas de um total de 10 transmissões. Por fim, a última coluna apresenta o resultado do conjunto de transmissões, sendo “*SUCCESS*” para bem sucedidos, “*ACCEPTABLE*” para aceitáveis e “*FAILED*” para falhos.

Figura 23 – Resultado gerado pelo Código F.3

```
1873
01k01k > 0100B > 04.0cm
0500Hz 10 00 00 10/10 SUCCESS
1000Hz 10 00 00 10/10 SUCCESS
1500Hz 10 00 00 10/10 SUCCESS
2000Hz 10 00 00 10/10 SUCCESS
2500Hz 10 00 00 10/10 SUCCESS
3000Hz 10 00 00 10/10 SUCCESS
3500Hz 10 00 00 10/10 SUCCESS
4000Hz 10 00 00 10/10 SUCCESS
4500Hz 10 00 00 10/10 SUCCESS
5000Hz 09 01 00 10/10 ACCEPTABLE
5500Hz 10 00 00 10/10 SUCCESS
6000Hz 09 01 00 10/10 ACCEPTABLE
6500Hz 00 00 10 00/10 FAILED
7000Hz 00 00 10 00/10 FAILED
```

Fonte: Autor.

3.5.7 Função *deleteJsons()*

A função mais simples é a *deleteJsons()*. A função dela, como sugere, é simplesmente apagar qualquer arquivo com nome de “*result.json*” presente em qualquer diretório da árvore de diretórios a partir do diretório raiz.

4 RESULTADOS E DISCUSSÕES

Os resultados aqui são dispostos em um primeiro nível com relação à configuração do divisor de tensão, o qual tem sua tensão de saída utilizada como tensão de referência no comparador, sendo $R_1 = 1 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$, $R_1 = 10 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$ ou $R_1 = 50 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$.

Em um segundo nível há a quantidade de *bytes*, 100 *bytes*, 500 *bytes* ou 1000 *bytes*.

Dentro do segundo nível, são exibidos os dados com relação a quantidade de transmissões limpas, sujas e falhas e a consideração final com relação ao conjunto de 10 transmissões (ver [Seção 3.5.2](#)).

Nas imagens, e ao longo do texto, será levado em consideração a frequência de transmissão de dados. Considerando que apenas um *bit* é amostrado por ciclo e com base na [Equação \(1\)](#), temos que o *bit rate* é igual à frequência.

Toda a coleta de dados foi realizada em um ambiente escuro.

4.1 DIVISOR DE TENSÃO COM $R_1 = 1 \text{ k}\Omega$ E $R_2 = 1 \text{ k}\Omega$

Ao longo desta seção, a configuração do divisor de tensão, utilizado para a tensão de referência do comparador feito através do amplificador operacional, será $R_1 = 1 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$. Com uma entrada de 5 V no divisor, a tensão de referência foi de 2.5 V. Devido à tensão de referência, a distância máxima de transmissão para a configuração foi de 5.0 cm, sendo que os dados eram sempre baixos na saída do comparador para distâncias maiores que 5.0 cm, e assim, o processamento do receptor mantinha-se sempre em estado ocioso.

4.1.1 Carga de dados de 100 *bytes*

Os resultados das transmissões, com base na distância e na frequência, para uma carga de dados de 100 *bytes*, são mostrados na [Figura 24](#).

Percebe-se que as frequências mais estáveis foram de 500 Hz, 1000 Hz e 1500 Hz, em que tiveram o conjunto de transmissões bem sucedido (10 transmissões limpas) para todas as distâncias testadas.

A distância em que houve um conjunto de transmissão bem sucedido com a maior frequência foi de 3.5 cm, ou seja, uma distância que representa 70% da distância máxima de transmissão (5.0 cm). Além disso, essa foi a distância com maior

quantidade de conjuntos de transmissões bem sucedidos ao longo da variação de frequência, sendo:

- 12 bem sucedidos (de 500 Hz até 6000 Hz);
- 1 aceitável (6500 Hz);
- 1 falho (7000 Hz).

A distância em que houve o primeiro aumento na frequência de transmissão, para um conjunto não falho, foi de 1.5 cm (30% da distância máxima de transmissão). Considerando a distância anterior (1.0 cm), a frequência foi de 3500 Hz para 4000 Hz.

Considerando a distância máxima, observou-se o pior desempenho do sistema, em que as frequências mais estáveis (de 500 Hz a 1500 Hz) mostraram-se viáveis, havendo ainda um conjunto aceitável para 2000 Hz.

Figura 24 – Resultados para divisor de tensão feito com $R_1 = 1 \text{ k}\Omega$ e $R_2 = 1 \text{k}\Omega$ e carga de dados de 100 bytes

FREQUÊNCIA (Hz)	7000							L: 00 S: 00 F: 10	L: 00 S: 00 F: 10		
	6500							L: 04 S: 06 F: 00	L: 00 S: 00 F: 10		
	6000							L: 10 S: 00 F: 00	L: 01 S: 09 F: 00		
	5500							L: 00 S: 00 F: 10	L: 10 S: 00 F: 00	L: 10 S: 00 F: 00	
	5000							L: 00 S: 00 F: 10	L: 01 S: 09 F: 00	L: 10 S: 00 F: 00	L: 01 S: 09 F: 00
	4500				L: 00 S: 00 F: 10	L: 00 S: 00 F: 10	L: 01 S: 09 F: 00	L: 10 S: 00 F: 00	L: 10 S: 00 F: 00	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10
	4000	L: 00 S: 00 F: 10	L: 00 S: 00 F: 10	L: 00 S: 00 F: 10	L: 00 S: 10 F: 00	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10				
	3500	L: 10 S: 00 F: 00									
	3000	L: 10 S: 00 F: 00									
	2500	L: 10 S: 00 F: 00									
	2000	L: 10 S: 00 F: 00									
	1500	L: 10 S: 00 F: 00									
	1000	L: 10 S: 00 F: 00									
	0500	L: 10 S: 00 F: 00									
1k Ω + 1k Ω 100 Bytes	00.0	00.5	01.0	01.5	02.0	02.5	03.0	03.5	04.0	04.5	05.0

DISTÂNCIA (cm)

L: Quantidade de transmissões limpas S: Quantidade de transmissões sujas F: Quantidade de transmissões falhas

Transmissões limpas = 10 Transmissões limpas < 10
 Transmissões falhas < 02

Frequência não testada para a distância correspondente devido a uma falha (visível) de transmissão para a frequência anterior

Fonte: Autor.

4.1.2 Carga de dados de 500 bytes

A figura [Figura 25](#) apresenta os resultados para uma carga de dados de 500 bytes para a configuração atual.

Assim como para a carga de 100 bytes, na [Seção 4.1.1](#), as frequências mais estáveis foram de 500 Hz, 1000 Hz e 1500 Hz, em que tiveram o conjunto de transmissões bem sucedido (10 transmissões limpas) para todas as distâncias testadas, exceto para a distância de 1.0 cm, em que houve um conjunto aceitável para 1000 Hz.

O conjunto bem sucedido com maior frequência foi obtido em 3.5 cm (70.0% da distância máxima de transmissão). Foi também a distância com maior quantidade de conjuntos de transmissões bem sucedidos ao longo da variação de frequência, sendo:

- 12 bem sucedidos (de 500 Hz até 6000 Hz);
- 1 aceitável (6500 Hz);
- 1 falho (7000 Hz).

A frequência de transmissão iniciou um aumento em 1.5 cm (30% da distância máxima de transmissão), em que a frequência foi de 3500 Hz para 4000 Hz, comparando com a distância de 1.0 cm

Em relação à distância máxima, observou-se novamente o pior desempenho do sistema, em que apenas as frequências mais estáveis (de 500 Hz à 1500 Hz) mostraram-se viáveis.

Figura 25 – Resultados para divisor de tensão feito com $R_1 = 1 \text{ k}\Omega$ e $R_2 = 1 \text{k}\Omega$ e carga de dados de 500 bytes

FREQUÊNCIA (Hz)	7000						L: 00 S: 00 F: 10	L: 00 S: 00 F: 10				
	6500						L: 00 S: 00 F: 10	L: 06 S: 04 F: 00	L: 00 S: 00 F: 10			
	6000						L: 00 S: 00 F: 10	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10			
	5500						L: 00 S: 10 F: 00	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10			
	5000						L: 00 S: 00 F: 10	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10			
	4500			L: 00 S: 00 F: 10	L: 00 S: 00 F: 10	L: 00 S: 10 F: 00	L: 10 S: 00 F: 00	L: 10 S: 00 F: 00	L: 10 S: 00 F: 10			
	4000	L: 00 S: 00 F: 10	L: 00 S: 00 F: 10	L: 00 S: 00 F: 10	L: 10 S: 00 F: 00	L: 10 S: 00 F: 10						
	3500	L: 02 S: 08 F: 00	L: 10 S: 00 F: 00	L: 09 S: 01 F: 00	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10						
	3000	L: 10 S: 00 F: 00	L: 01 S: 09 F: 00									
	2500	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10									
	2000	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10									
	1500	L: 10 S: 00 F: 00										
	1000	L: 10 S: 00 F: 00										
	0500	L: 10 S: 00 F: 00										
	1k Ω + 1k Ω 500 Bytes	00.0	00.5	01.0	01.5	02.0	02.5	03.0	03.5	04.0	04.5	05.0
	DISTÂNCIA (cm)											

L: Quantidade de transmissões limpas S: Quantidade de transmissões sujas F: Quantidade de transmissões falhas

Transmissões limpas = 10 Transmissões limpas < 10
Transmissões falhas < 02 Transmissões falhas ≥ 2

Frequência não testada para a distância correspondente devido a uma falha (visível) de transmissão para a frequência anterior

Fonte: Autor.

4.1.3 Carga de dados de 1000 bytes

Por fim, para a configuração do divisor de tensão feita com $R_1 = 1 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$, os resultados para uma carga de dados de 1000 *bytes* são mostrados na Figura 26.

Para as frequências mais estáveis, essa carga de *bytes* segue o mesmo resultado para as duas cargas menores para essa configuração do divisor de tensão, ou seja, as frequências mais estáveis foram de 500 Hz, 1000 Hz e 1500 Hz, em que tiveram o conjunto de transmissões bem sucedido (10 transmissões limpas) para todas as distâncias testadas.

Em 3.5 cm houve a maior frequência de transmissão com conjunto de transmissões bem sucedido, ou seja, uma distância que representa 70% da distância máxima de transmissão (5.0 cm). Além disso, essa foi a distância com maior quantidade de conjuntos de transmissões bem sucedidos ao longo da variação de frequência, sendo:

- 12 bem sucedidos (de 500 Hz até 6000 Hz);
- 2 aceitáveis (6500 Hz e 7000 Hz);
- 1 falho (7500 Hz).

Dessa vez, a distância em que a frequência de transmissão começou a aumentar foi igual 1.0 cm (20% da distância máxima de transmissão), considerando conjuntos não falhos. De 0.5 cm para 1.0 cm a frequência aumentou de 3500 Hz para 4.0 Hz.

Mais uma vez, observou-se o pior desempenho do sistema em distância máxima. Apenas as frequências mais estáveis (de 500 Hz à 1500 Hz) mostraram-se viáveis.

Figura 26 – Resultados para divisor de tensão feito com $R_1 = 1\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$ e carga de dados de 1000 bytes

FREQUÊNCIA (Hz)	7500							L: 00 S: 00 F: 10	L: 00 S: 00 F: 10		
	7000							L: 00 S: 10 F: 00	L: 00 S: 00 F: 10		
	6500							L: 00 S: 00 F: 10	L: 04 S: 06 F: 00	L: 00 S: 00 F: 10	
	6000							L: 00 S: 00 F: 10	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10	
	5500							L: 00 S: 10 F: 00	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10	
	5000							L: 00 S: 00 F: 10	L: 07 S: 00 F: 00	L: 10 S: 00 F: 00	L: 10
	4500		L: 00 S: 00 F: 10	L: 00 S: 00 F: 10	L: 00 S: 00 F: 10	L: 02 S: 08 F: 00	L: 10 S: 00 F: 00				
	4000	L: 00 S: 00 F: 10	L: 00 S: 00 F: 10	L: 00 S: 10 F: 00	L: 08 S: 02 F: 00	L: 10 S: 00 F: 10					
	3500	L: 08 S: 02 F: 00	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10							
	3000	L: 10 S: 00 F: 00	L: 00 S: 10 F: 00								
2500	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10									
2000	L: 10 S: 00 F: 00	L: 00 S: 00 F: 10									
1500	L: 10 S: 00 F: 00										
1000	L: 10 S: 00 F: 00										
0500	L: 10 S: 00 F: 00										
1kΩ + 1kΩ 1000 Bytes	00.0	00.5	01.0	01.5	02.0	02.5	03.0	03.5	04.0	04.5	05.0

DISTÂNCIA (cm)

L: Quantidade de transmissões limpas S: Quantidade de transmissões sujas F: Quantidade de transmissões falhas

 Transmissões limpas = 10 Transmissões limpas < 10
Transmissões falhas < 02 Transmissões falhas ≥ 2

 Frequência não testada para a distância correspondente devido a uma falha (visível) de transmissão para a frequência anterior

Fonte: Autor.

4.2 DIVISOR DE TENSÃO COM $R_1 = 10 \text{ k}\Omega$ E $R_2 = 1 \text{ k}\Omega$

Nesta seção, a configuração do divisor de tensão, utilizado para obter a tensão de referência do comparador, foi feita com $R_1 = 10 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$. Sendo a entrada de 5 V no divisor, a tensão de referência foi de 0.455 V. Devido a tensão de referência, a distância máxima de transmissão para a configuração foi de 14.5 cm. Distâncias maiores que 14.5 cm apresentaram valores sempre baixo na saída do comparador, mantendo o sistema de recepção ocioso.

4.2.1 Carga de dados de 100 *bytes*

A [Figura 27](#) apresenta os resultados das transmissões, com base na distância e na frequência, para uma carga de dados de 100 *bytes*.

As frequências mais estáveis foram de 500 Hz e 1000 Hz, sendo que apresentaram, com relação ao conjunto de 10 transmissões:

- 500 Hz: 29 conjuntos bem sucedidos e 1 conjunto falho de 30 conjuntos, sendo o conjunto falho na distância máxima (14.5 cm);
- 1000 Hz: 28 conjuntos bem sucedidos e 2 conjuntos falhos de 30 conjuntos, sendo os conjuntos falhos na distância anterior à máxima e na máxima (14.0 cm e 14.5 cm).

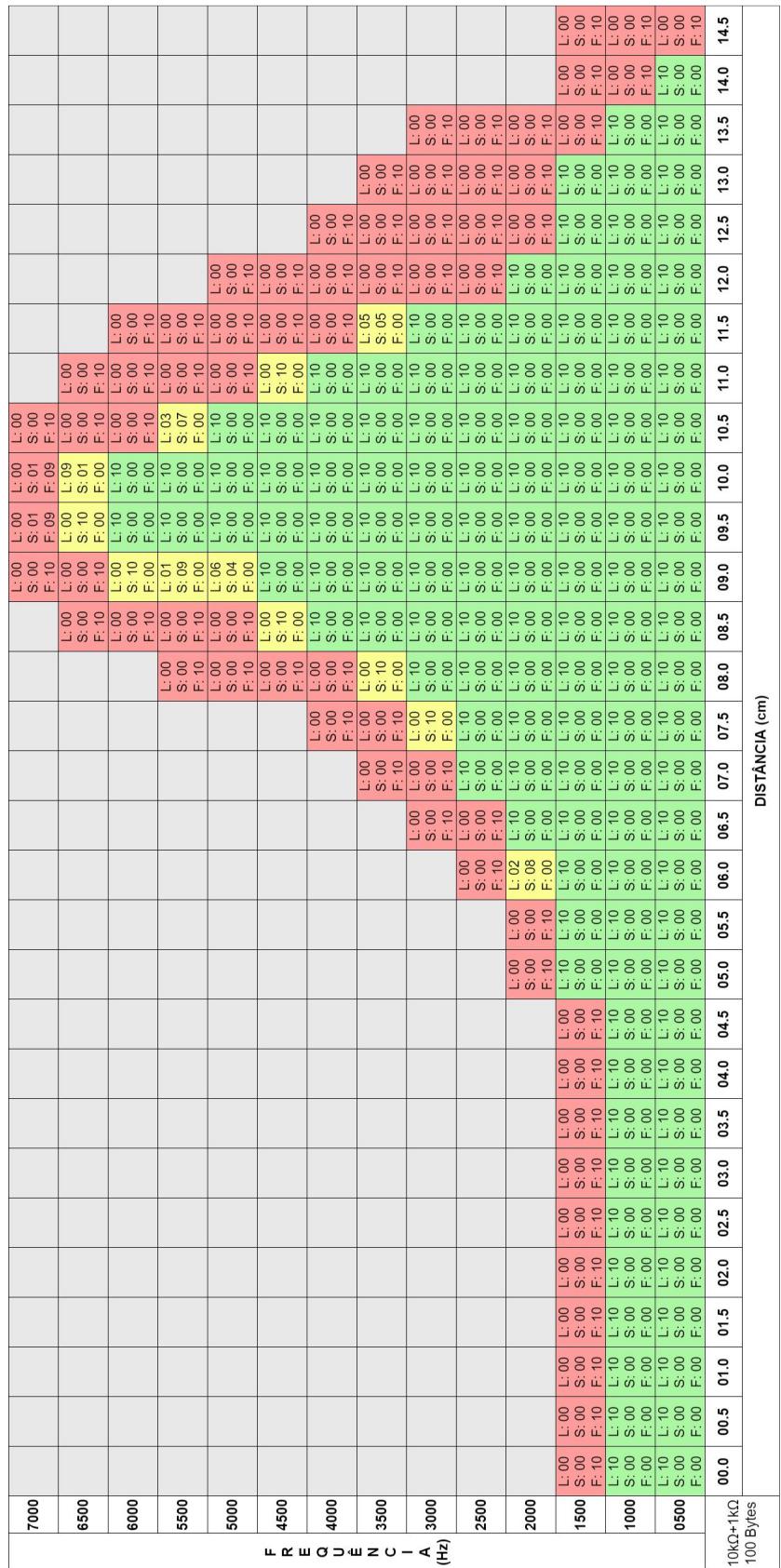
A maior frequência com um conjunto de transmissão bem sucedido ocorreu em 9.5 cm e 10.0 cm, ou seja, distâncias que representam 65.5% e 69.0% da distância máxima de transmissão. Essas também foram as distâncias com maior quantidade de conjuntos de transmissões bem sucedidos ao longo da variação de frequência, sendo:

- 12 bem sucedidos (de 500 Hz até 6000 Hz);
- 1 aceitável (6500 Hz);
- 1 falho (7500 Hz).

A máxima frequência de transmissão foi de 1000 Hz para distâncias de 0.0 cm até 4.5 cm (31% da distância máxima), sendo todos conjuntos bem sucedidos. Ainda nesse intervalo de distância, os conjuntos para 1500 Hz foram falhos. A frequência de 1000 Hz corresponde a 16.67% da frequência máxima atingida pelo sistema (6000 Hz) em todas as configurações possíveis de distância e frequência, para um conjunto não falho.

Não diferente dos casos anteriores na configuração da [Seção 4.1](#), em distância máxima o sistema teve o pior desempenho, mas, nesse caso, todas as tentativas de transmissão falharam.

Figura 27 – Resultados para divisor de tensão feito com $R_1 = 10 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$ e carga de dados de 100 bytes



Fonte: Autor.

L: Quantidade de transmissões limpas

S: Quantidade de transmissões sujas

 Transmissões limpas = 10

 Transmissões limpas < 10

 Transmissões falhas < 2

 Transmissões falhas > 2

F: Quantidade de transmissões falhas

F: Frequência não testada para a distância correspondente devido a uma falha (visível) de transmissão para a frequência anterior

4.2.2 Carga de dados de 500 bytes

Os resultados das transmissões, com base na distância e na frequência, para uma carga de dados de 500 bytes, são mostrados na [Figura 28](#).

Para as frequências de 500 Hz e 1000 Hz, obteve-se a melhor estabilidade. Com relação ao conjunto de 10 transmissões dessas frequências:

- 500 Hz: 30 conjuntos bem sucedidos de 30 conjuntos;
- 1000 Hz: 29 conjuntos bem sucedidos e 1 conjunto falho de 30 conjuntos, sendo o conjunto falho na distância máxima (14.5 cm).

Em 10.0 cm (69.0% da distância máxima de transmissão), obteve-se um conjunto de transmissões limpas com a maior frequência. Essa foi a distância com maior quantidade de conjuntos de transmissões bem sucedidos ao longo da variação de frequência, sendo:

- 12 bem sucedidos (de 500 Hz até 6000 Hz);
- 2 aceitáveis (6500 Hz e 7000 Hz);
- 2 falhos (7500 Hz e 8000 Hz).

De 0.0 cm até 5.0 cm (35.0% da distância máxima), os resultados finais foram similares para os conjuntos de transmissões, máxima frequência de transmissão de 1000 Hz para bem sucedidos e 1500 Hz para falhos. Assim, a frequência máxima de transmissão, nessa faixa de distâncias (0.0 cm até 5.0 cm), foi de 1000 Hz, 16.67% da frequência máxima atingida pelo sistema em todas as configurações possíveis de distância e frequência, considerando um conjunto bem sucedido.

O pior desempenho do sistema ocorreu em máxima distância, em que apenas a frequência de 500 Hz demonstrou-se viável.

Figura 28 – Resultados para divisor de tensão feito com $R_1 = 10 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$ e carga de dados de 500 bytes

FREQUÊNCIA (Hz)	DISTÂNCIA (cm)	L: Quantidade de transmissões limpas		S: Quantidade de transmissões sujas		F: Quantidade de transmissões falhas
		Transmissões limpas = 10	Transmissões limpas < 10	Transmissões sujas = 2	Transmissões sujas < 2	
8000	L:00	L:00	S:00	S:00	S:00	F:10
8000	F:10	F:10	F:10	F:10	F:10	F:10
7500	L:00	L:00	S:00	S:00	S:00	F:10
7500	F:08	F:08	F:10	F:10	F:10	F:10
7000	L:00	L:00	L:00	L:00	L:00	L:00
7000	F:10	F:10	F:00	F:00	F:00	F:10
6500	L:00	L:00	L:06	L:00	L:00	L:00
6500	F:10	F:10	F:04	S:00	S:00	F:10
6000	L:00	L:00	L:10	L:00	L:00	L:00
6000	F:10	F:10	F:00	F:00	F:00	F:10
5500	L:00	L:00	L:10	L:00	L:00	L:00
5500	F:10	F:10	F:00	F:00	F:00	F:10
5000	L:00	L:00	L:10	L:00	L:00	L:00
5000	F:10	F:10	F:00	F:00	F:00	F:10
4500	L:00	L:04	L:10	L:00	L:00	L:00
4500	F:10	F:06	F:00	S:00	S:00	F:10
4000	L:00	L:00	L:10	L:00	L:00	L:00
4000	F:10	F:08	F:00	S:00	S:00	F:10
3500	L:00	L:00	L:10	L:00	L:00	L:00
3500	F:10	F:10	F:00	F:00	F:00	F:10
3000	L:00	L:00	L:10	L:00	L:00	L:00
3000	F:10	F:10	F:00	F:00	F:00	F:10
2500	L:00	L:00	L:10	L:00	L:00	L:00
2500	F:10	F:10	F:00	F:00	F:00	F:10
2000	L:00	L:00	L:10	L:00	L:00	L:00
2000	F:10	F:10	F:00	F:00	F:00	F:10
1500	L:00	L:00	L:10	L:00	L:00	L:00
1500	F:10	F:10	F:10	F:10	F:10	F:10
1000	L:10	L:10	L:10	L:10	L:10	L:10
1000	F:00	F:00	F:00	F:00	F:00	F:00
0500	L:10	L:10	L:10	L:10	L:10	L:10
0500	F:00	F:00	F:00	F:00	F:00	F:00
000	00.0	00.5	01.0	01.5	02.0	02.5
500 Bytes	03.0	03.5	04.0	04.5	05.0	05.5
10kΩ+1kΩ	06.0	06.5	07.0	07.5	08.0	08.5
500 Bytes	09.0	09.5	10.0	10.5	11.0	11.5
	12.0	12.5	13.0	13.5	14.0	14.5

Fonte: Autor.

L: Quantidade de transmissões limpas S: Quantidade de transmissões sujas
 Transmissões limpas = 10 Transmissões limpas < 10 Transmissões sujas = 2 Transmissões sujas < 2
 F: Quantidade de transmissões falhas
 Frequência não testada para a distância correspondente devido a uma falha (visível) de transmissão para a frequência anterior

4.2.3 Carga de dados de 1000 bytes

Para a última carga dessa configuração de divisor de tensão, 1000 *bytes*, os resultados são mostrados na [Figura 29](#).

Temos que as frequências mais estáveis foram de 500 Hz e 1000 Hz, sendo que apresentaram, com relação ao conjunto de 10 transmissões:

- 500 Hz: 30 conjuntos bem sucedidos de 30 conjuntos;
- 1000 Hz: 28 conjuntos bem sucedidos e 2 conjuntos falhos de 30 conjuntos, sendo os conjuntos falhos na distância anterior a máxima e na máxima (14.0 cm e 14.5 cm).

A distância em que houve um conjunto de transmissão bem sucedido com a maior frequência foi de 10.0 cm (69.0% da distância máxima de transmissão). Além disso, essa foi a distância com maior quantidade de conjuntos de transmissões bem sucedidos ao longo da variação de frequência. Os conjuntos, nessa distância, foram:

- 11 bem sucedidos (de 500 Hz até 5000 Hz e 6000 Hz);
- 3 aceitáveis (5500 Hz, 6500 Hz e 7000 Hz);
- 1 falho (7500 Hz).

Os resultados finais foram similares para os conjuntos de transmissões no intervalo de 0.0 cm até 4.5 cm. A máxima frequência para bem sucedidos foi de 1000 Hz e 1500 Hz para falhos. Considerando conjuntos bem sucedidos, a máxima frequência de transmissão nessa faixa de distâncias foi de 16.67% da frequência máxima atingida pelo sistema em todas as configurações possíveis de distância e frequência.

Apenas a frequência de 500 Hz demonstrou-se viável em máxima distância, sendo o pior desempenho.

Figura 29 – Resultados para divisor de tensão feito com $R_1 = 10 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$ e carga de dados de 1000 bytes

		DISTÂNCIA (cm)									
		L: Quantidade de transmissões limpas					S: Quantidade de transmissões sujas				
		Transmissões limpas < 10					Transmissões sujas				
		Transmissões falhas > 2					Transmissões falhas > 2				
8000	L:00	S:00	L:00	S:00	L:00	S:00	L:00	S:00	L:00	S:00	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
7500	L:00	S:02	L:00	S:02	L:00	S:02	L:00	S:02	L:00	S:02	L:00
	F:08	F:08	F:08	F:08	F:08	F:08	F:08	F:08	F:08	F:08	F:08
7000	L:00	S:04	L:00	S:04	L:00	S:04	L:00	S:04	L:00	S:04	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
6500	L:00	S:06	L:01	S:06	L:00	S:06	L:00	S:06	L:00	S:06	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
6000	L:00	S:08	L:00	S:08	L:00	S:08	L:00	S:08	L:00	S:08	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
5500	L:00	S:09	L:00	S:09	L:00	S:09	L:00	S:09	L:00	S:09	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
5000	L:00	S:10	L:00	S:10	L:00	S:10	L:00	S:10	L:00	S:10	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
4500	L:00	S:09	L:01	S:09	L:00	S:09	L:00	S:09	L:00	S:09	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
4000	L:00	S:10	L:00	S:10	L:00	S:10	L:00	S:10	L:00	S:10	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
3500	L:00	S:09	L:00	S:09	L:00	S:09	L:00	S:09	L:00	S:09	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
3000	L:00	S:08	L:00	S:08	L:00	S:08	L:00	S:08	L:00	S:08	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
2500	L:00	S:08	L:00	S:08	L:00	S:08	L:00	S:08	L:00	S:08	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
2000	L:00	S:09	L:00	S:09	L:00	S:09	L:00	S:09	L:00	S:09	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
1500	L:00	S:08	L:00	S:08	L:00	S:08	L:00	S:08	L:00	S:08	L:00
	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10	F:10
1000	L:10	S:00	L:10	S:00	L:10	S:00	L:10	S:00	L:10	S:00	L:10
	F:00	F:00	F:00	F:00	F:00	F:00	F:00	F:00	F:00	F:00	F:00
0500	L:10	S:00	L:10	S:00	L:10	S:00	L:10	S:00	L:10	S:00	L:10
	F:00	F:00	F:00	F:00	F:00	F:00	F:00	F:00	F:00	F:00	F:00
000	00.0	00.5	01.0	01.5	02.0	02.5	03.0	03.5	04.0	04.5	05.0
10kΩ+1kΩ	0.00	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50
1000 Bytes	0.00	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50

Fonte: Autor.

L: Quantidade de transmissões limpas
■ Transmissões limpas = 10
■ Transmissões limpas < 10
■ Transmissões sujas
■ Transmissões sujas

F: Quantidade de transmissões falhas
■ Freqüência não testada para a distância correspondente devido a uma falha (visível) de transmissão para a freqüência anterior

4.3 DIVISOR DE TENSÃO COM $R_1 = 50\text{ k}\Omega$ E $R_2 = 1\text{ k}\Omega$

Durante esta seção, será utilizado para o divisor de tensão $R_1 = 50\text{ k}\Omega + R_2 = 1\text{ k}\Omega$. Com uma entrada de 5 V no divisor, a tensão de referência foi de 0.098 V, utilizada no comparador. Devido a tensão de referência, a distância máxima de transmissão para a configuração foi em torno de 30 cm. Diferentemente das configurações anteriores, havia envio e recebimento de dados para distâncias maiores, porém o LED emissor e o fototransistor receptor precisam estar em um alinhamento perfeito, principalmente devido à potência do LED. Desse modo, limitou-se a distância máxima de transmissão para captura de dados para essa tensão de referência a 30 cm. Porém, para a análise será utilizada a frequência máxima em que observou-se uma comunicação possível que gerasse um conjunto bem sucedido, a qual foi de 34.0 cm

4.3.1 Carga de dados de 100 *bytes*

A [Figura 30](#) contém os resultados das transmissões, com base na distância e na frequência, para uma carga de dados de 100 *bytes*.

A frequência mais estável foi de 500 Hz, sendo que apresentou, com relação ao conjunto de 10 transmissões:

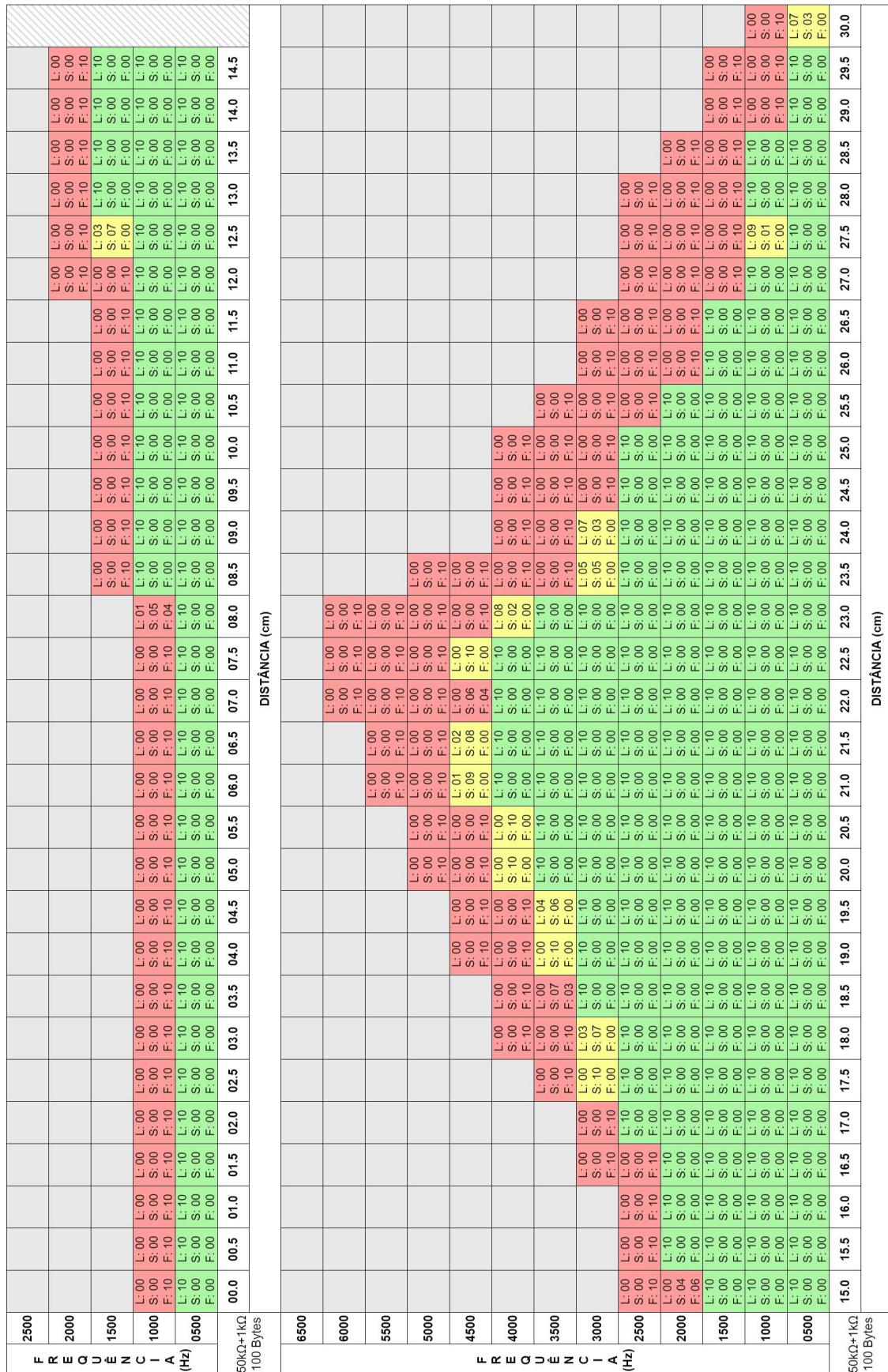
- 500 Hz: 60 conjuntos bem sucedidos e 1 conjunto aceitável de 61 conjuntos, sendo o conjunto aceitável na distância máxima (30.0 cm);

As distâncias de 21.0 cm até 22.5 cm apresentaram um conjunto de transmissão bem sucedido com a maior frequência. Essas distâncias representam de 61.8% a 66.2% da distância máxima de transmissão. Foram as distâncias com maior quantidade de conjuntos de transmissões bem sucedidos ao longo da variação de frequência, sendo 8 conjuntos, de 500 Hz até 4000 Hz.

Já as distâncias de 0.0 cm a 8.0 cm (23.6% da distância máxima) apresentaram resultados finais iguais para os conjuntos de transmissões, máxima frequência de 500 Hz para bem sucedidos e 1000 Hz para falhos. Assim, uma comunicação bem sucedida, nesse intervalo, ocorreu a no máximo 12.5% da frequência máxima do sistema, considerando todas as possibilidades de distância e frequência.

Em distância máxima, observou-se o pior desempenho do sistema, em que apenas a frequência de 500 Hz demonstrou-se viável, sem ser um conjunto bem sucedido, mas sim viável.

Figura 30 – Resultados para divisor de tensão feito com $R_1 = 50 \text{ k}\Omega$ e $R_2 = 1 \text{ k}\Omega$ e carga de dados de 100 bytes



Fonte: Autor.

L: Quantidade de transmissões limpas
S: Quantidade de transmissões sujas

Transmissões limpas = 10

Transmissões sujas < 10

Transmissões falhas < 2

Transmissões falhas ≥ 2

F: Frequência de transmissões falhas

F: Frequência não testada para a distância correspondente devido a uma falha (visível) de transmissão para a frequência anterior

4.3.2 Carga de dados de 500 bytes

Para as distâncias e frequências, os resultados das transmissões com uma carga de 500 *bytes* são mostrados na [Figura 31](#).

Novamente, para essa configuração do divisor de tensão, a frequência mais estável foi de 500 Hz, sendo que apresentou, com relação ao conjunto de 10 transmissões:

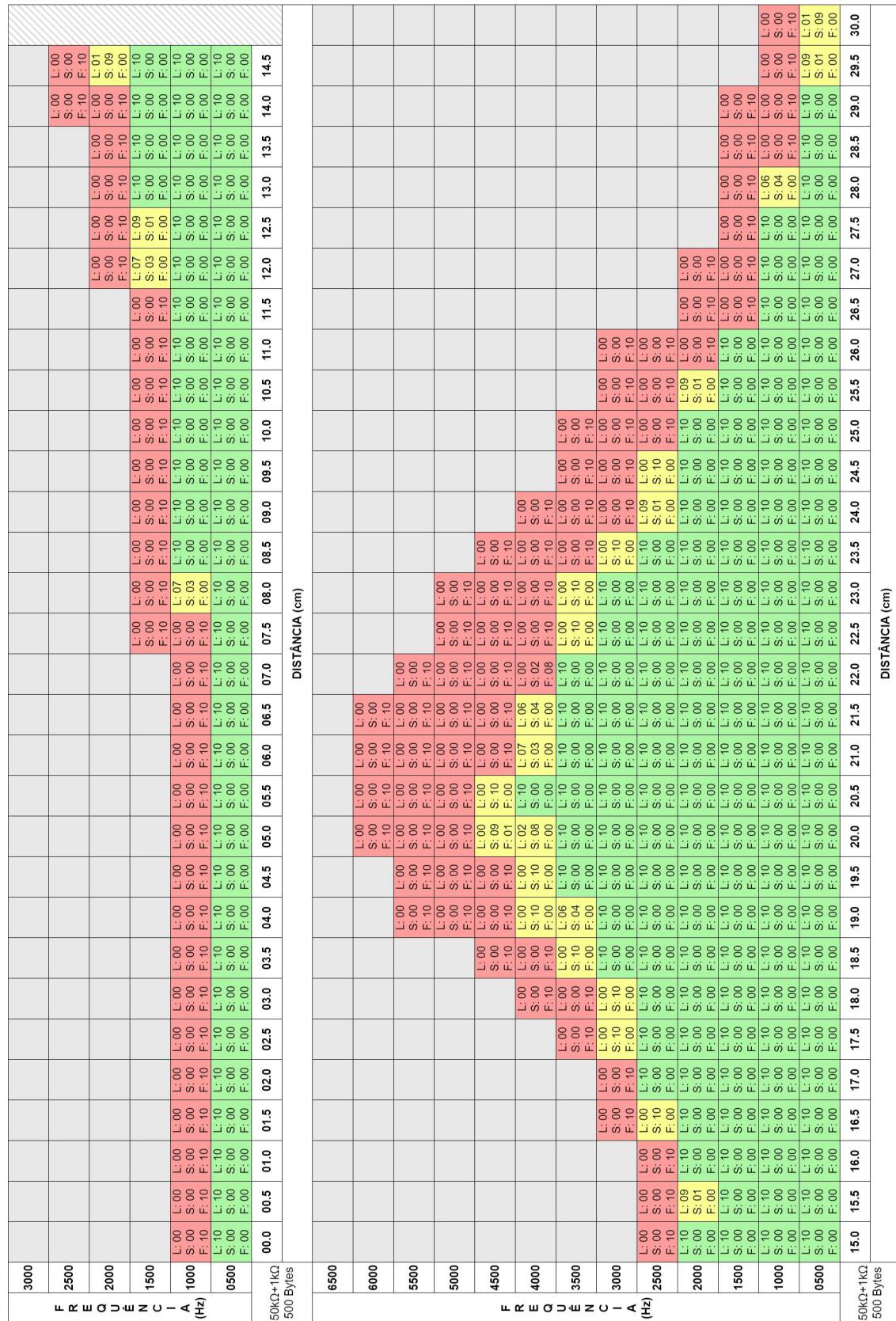
- 500 Hz: 59 conjuntos bem sucedidos e 2 conjunto aceitáveis de 61 conjuntos, sendo os conjuntos aceitáveis na distância anterior à máxima e na máxima (29.5 cm e 30.0 cm);

A distância em que houve um conjunto de transmissão bem sucedido com a maior frequência foi de 20.5 cm, ou seja, cerca de 60.3% da distância máxima de transmissão. Essa foi a distância com maior quantidade de conjuntos de transmissões bem sucedidos ao longo da variação de frequência, sendo 8 conjuntos, de 500 Hz até 4000 Hz.

Além disso, as distâncias de 0.0 cm até 7.5 cm, 22.06% da distância máxima, apresentaram resultados finais iguais para os conjuntos de transmissões, máxima frequência de 500 Hz para bem sucedidos e 1000 Hz para falhos. Dessa forma, uma comunicação bem sucedida ocorreu a no máximo 12.5% da frequência máxima atingida pelo sistema, nessa faixa de distância e para essa configuração.

Em distância máxima e anterior à máxima, observou-se o pior desempenho do sistema, em que apenas a frequência de 500 Hz foi viável, sendo um conjunto aceitável.

Figura 31 – Resultados para divisor de tensão feito com $R_1 = 50\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$ e carga de dados de 500 bytes



Fonte: Autor.

4.3.3 Carga de dados de 1000 bytes

Por fim, os resultados das transmissões para uma carga de dados de 1000 bytes, com base na distância e na frequência, são mostrados na [Figura 32](#).

Mais uma vez a frequência mais estável foi de 500 Hz, sendo que apresentou, com relação ao conjunto de 10 transmissões:

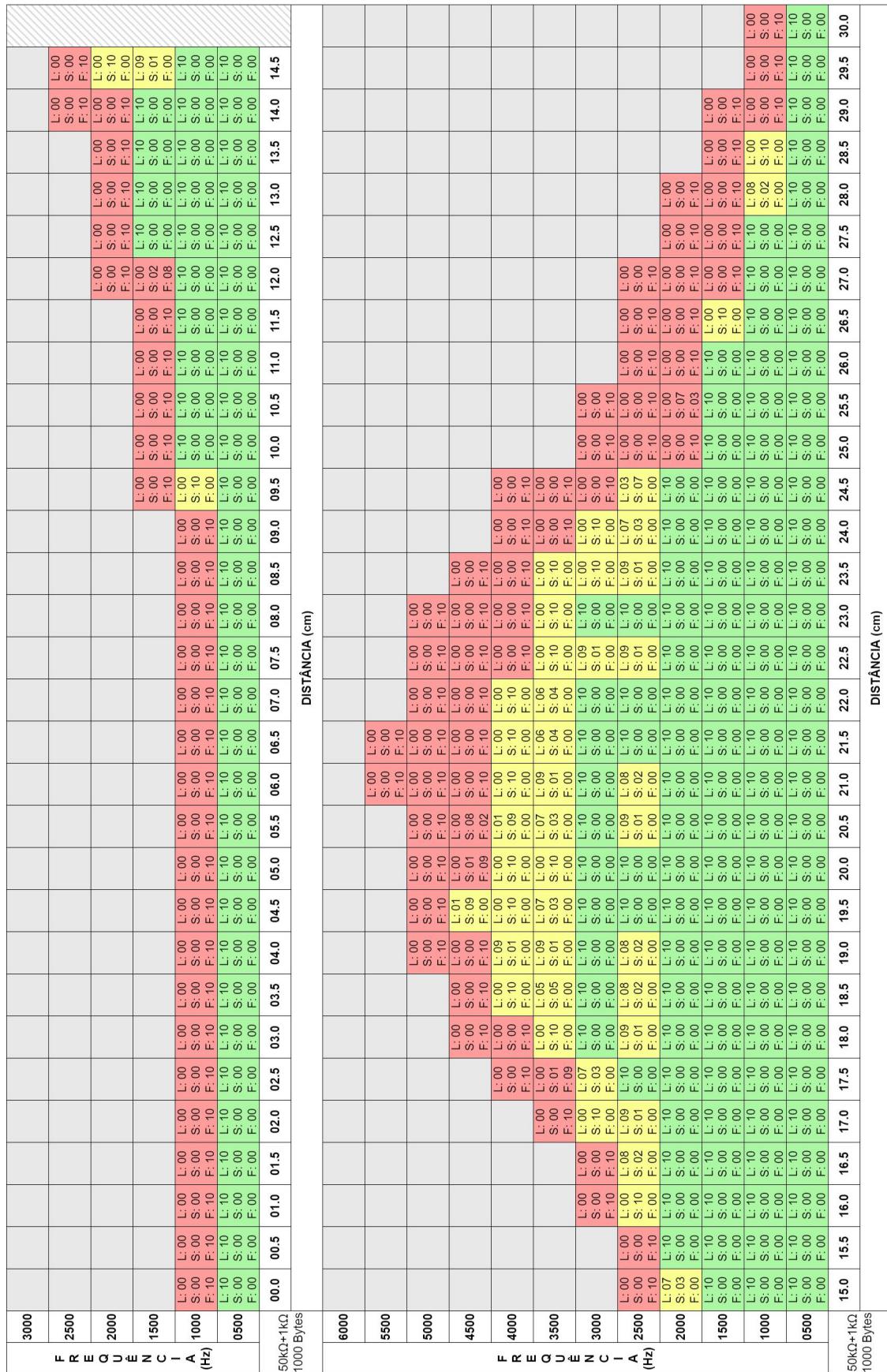
- 500 Hz: 61 conjuntos bem sucedidos de 61 conjuntos;

As distâncias em que houveram um conjunto de transmissão bem sucedido com a maior frequência foram de 18.0 cm até 22.0 cm e 23.0 cm, ou seja, de 52.9% até 67.7% da distância máxima de transmissão. As distâncias com maior quantidade de conjuntos de transmissões bem sucedidos ao longo da variação de frequência foram de 19.5 cm, 20.0 cm, 21.5 cm, 22.0 cm e 23.0 cm, sendo 6 conjuntos, de 500 Hz até 3000 Hz.

Temos ainda que as distâncias de 0.0 cm até 9.0 cm, 26.5% da distância máxima de transmissão, apresentaram resultados finais iguais para os conjuntos de transmissões, máxima frequência de 500 Hz para bem sucedidos e 1000 Hz para falhos. Portanto, uma comunicação bem sucedida ocorreu a no máximo 12.5% da frequência máxima atingida pelo sistema com conjunto bem sucedido, nessa faixa de distância.

De 29.0 cm até 30.0 cm, o pior desempenho do sistema foi observada, em que apenas a frequência de 500 Hz demonstrou-se viável, sendo um conjunto bem sucedido.

Figura 32 – Resultados para divisor de tensão feito com $R_1 = 50\text{ k}\Omega$ e $R_2 = 1\text{ k}\Omega$ e carga de dados de 500 bytes



Fonte: Autor.

L: Quantidade de transmissões limpas
Transmissões limpas = 10 Transmissões limpas < 10
Transmissões falhas > 2 Transmissões falhas > 2

S: Quantidade de transmissões sujas
Transmissões sujas < 10 Transmissões sujas < 10
Transmissões sujas > 2 Transmissões sujas > 2

F: Frequência de transmissões falhas
Frequência não testada para a distância correspondente devido a uma falha (visível) de transmissão para a frequência anterior Frequência não testada para a distância correspondente devido a uma falha (visível) de transmissão para a frequência anterior

4.4 DISCUSSÃO SOBRE OS RESULTADOS

Os resultados nesta seção serão apresentados de forma resumida com relação aos resultados coletados e mencionados nas seções anteriores deste capítulo, [Capítulo 4](#).

Com relação às frequências mais estáveis ao longo das distâncias, observou-se que frequências mais baixas apresentam-se como melhor opção. Além disso, com a diminuição da tensão de referência do comparador, a faixa de frequências estáveis tende a diminuir. Como visto, a transmissão foi estável para a faixa de 500 Hz até 1500 Hz para a tensão de referência de 2.5 V. Para a tensão de referência de 0.455 V, a faixa de frequência estável foi de 500 Hz e 1000 Hz. Enquanto para a última configuração, com tensão de referência de 0.098 V, apenas 500 Hz demonstrou-se estável ao longo das distâncias analisadas. Também temos que, com a diminuição da tensão de referência no comparador houve um aumento da distância de transmissão máxima;

Já a maior frequência com um conjunto de transmissões bem sucedido ou aceitável aconteceu após metade da distância máxima de transmissão, entre 50.0% e 70.0%. Além disso, essa frequência máxima diminui na última tensão de referência do comparador, diminuindo de 6000 Hz a 7000 Hz nas duas maiores configurações de tensão de referência para 3000 Hz à 4500 Hz. Até o momento, não foi estudado o motivo das maiores frequências ocorrerem nessa faixa de porcentagem (de 50.0% até 70.0%) com relação à distância máxima.

O pior desempenho sempre ocorreu próximo às distâncias máximas de transmissão, mais especificamente em distâncias acima dos 85% da distância máxima.

Outro ponto é com relação à primeira distância em que há um aumento da frequência máxima de transmissão, momento em que também há um aumento da frequência de transmissão em que ocorre o primeiro conjunto falho. Em sua maioria, o limite da frequência máxima de transmissão tendeu a aumentar após a faixa de 20% a 35% da distância máxima.

Foi também observado que a diminuição da tensão de referência no comparador permitiu uma maior distância de transmissão. Isso era o esperado, já que a medida em que há o aumento da distância entre o transmissor e o receptor, há também a diminuição da tensão de saída do módulo TEMT6000 devido a menor intensidade de luz incidida no fototransistor. Como a saída do módulo é conectada à entrada a ser comparada (entrada não inversora) com a tensão de referência do comparador (entrada inversora), se esse sinal estiver sempre abaixo do nível do comparador, não

ocorre qualquer variação na saída.

Com base nos resultados observados, a carga de transmissão não se demonstrou um fator impactante na qualidade, diferentemente das outras variáveis, tensão de referência, distância e frequência de transmissão, as quais possuem grande impacto acerca da qualidade da transmissão.

4.5 COMPARAÇÃO COM OUTROS TRABALHOS

No trabalho “Data Transfer Using LED Light (Li-Fi)”, encontrado no repositório de projetos Arduino ([SAADSAIF0333, 2020](#)), uma implementação mais simples de comunicação por luz visível é implementada. Nesse projeto, é utilizado um LDR como receptor e funções *delay()* ao longo do código, os quais fatores limitantes, tanto na frequência máxima de transmissão quanto na propagação de atraso ao longo dos ciclos, fato que pode provocar uma perda de sincronização em longas transmissões.

Outro trabalho utilizando Arduino e VLC, [Visible Light Communication “VLC” or LIFI Project using Arduino \(FAHAD, 2020\)](#), também apresenta limitações de implementação semelhantes ao citado no parágrafo acima, além de utilizar funções embutidas de escrita nos pinos, aumentando ainda mais o custo de CPU para a transmissão de um dado.

Já no projeto descrito no texto “Visible light communication (Sending data through light)” ([HAMED; ODEH; AFANEH, 2015](#)), a implementação feita apresentou uma limitação de distância de transmissão, ficando limitado a no máximo 5 cm.

Por fim, outro projeto é descrito em “[DATA TRANSMISSION IN THE FORM OF LIGHT \(LIFI COMMUNICATION\) USING ARDUINO](#)” ([ELECTRONICS WORKSHOPS, 2020](#)) a limitação ocorreu também em questão de distância, em que houve problemas de transmissão a partir dos 15 cm.

Comparando com os projetos mencionados nessa seção, o trabalho descrito no [Capítulo 3](#) e no [Capítulo 4](#) aparente estar mais bem elaborado. Com relação a distância de transmissão, atualmente o limite ocorre com 34 cm, porém há o conhecimento de que esse valor pode ser aumentado aumentando a potência do LED. Com relação a taxa de transmissão, nenhum dos projetos citados nessa seção apresentaram os dados. Há uma demonstração em vídeo do projeto “Data Transfer Using LED Light (Li-Fi)” ([SAADSAIF0333, 2020](#)) em que a taxa de transferência é bem inferior a atingida na implementação dessa redação.

5 CONCLUSÃO E CONSIDERAÇÕES FINAIS

O projeto teve como objetivo inicial estabelecer um sistema de comunicação por luz visível utilizando plataformas e componentes periféricos acessíveis. Dessa forma, a plataforma de desenvolvimento utilizada foi a Arduino.

Uma vez estabelecida a comunicação, o objetivo foi realizar uma análise acerca de possíveis variáveis capazes de impactar a qualidade da transmissão dos dados. Mais especificamente, a tensão de referência utilizada no comparador do módulo receptor para determinar um *bit* com valor alto ou baixo, a carga de dados transmitida, a distância entre emissor e receptor e a frequência de transmissão de dados.

Com os devidos dados coletados, observou-se que a carga de transmissão não apresentou impacto significativo na qualidade de transmissão. Em contrapartida, a tensão de referência do comparador, a distância entre emissor e receptor e a frequência de transmissão demonstraram variáveis de grande impacto na transmissão. Como exemplo, observou-se uma maior estabilidade de transmissão de dados ao longo das distâncias ao utilizar frequências mais baixas e a frequência máxima de transmissão ocorreu sempre após 50% da máxima distância de transmissão, sendo a máxima distância controlada pela tensão de referência do comparador.

Em comparação com alguns projetos utilizando Arduino e VLC, o projeto desenvolvido aparenta uma melhor implementação com relação ao desempenho, principalmente por evitar utilizar funções que gerem atrasos de transmissão.

5.1 PROBLEMAS ENCONTRADOS

Deve-se ressaltar que toda a coleta dos dados foi realizada em um ambiente escuro, com o mínimo de iluminação possível no receptor. Em ambiente com outras fontes de luz, a qualidade de transmissão mostrou-se visivelmente pior. Esse é um ponto também levantado no artigo *Visible light communication: concepts, applications and challenges* ([MATHEUS, L. E. M. et al., 2019](#)).

Além disso, apesar da alimentação do amplificador operacional ter sido devidamente feita com 0 V na alimentação negativa e 5 V na alimentação positiva, a tensão de saída não foi superior à 4 V, sendo essa uma característica comum aos amplificadores operacionais. Como essa saída era suficiente para o desenvolvimento do projeto, o amplificador foi mantido.

Como levantado por em *Visible light communication: concepts, applications and challenges* ([MATHEUS, L. E. M. et al., 2019](#)), um dos problemas com relação ao VLC

é o *flickering* gerado por cada tipo de modulação, a oscilação ou cintilação do brilho da luz, sendo isso prejudicial à saúde dos olhos caso não seja mantido em um nível correto. Durante as transmissões, a alternância repentina de iluminação foi incômoda aos olhos.

A comunicação foi feita em via unidirecional, uma comunicação *simplex*. Ainda com relação ao artigo do parágrafo anterior, essa é uma das dificuldades em realizar a transmissão de dados no outro sentido, já que o retorno não pode interferir na comunicação. Radiofrequências foram utilizadas como alternativa para esse problema em específico.

5.2 SUGESTÕES PARA TRABALHOS FUTUROS

Um ponto de melhoria é o aperfeiçoamento de ambos os módulos. No caso do transmissor, aumentar a potência do LED provavelmente possibilitaria uma maior distância efetiva de transmissão, podendo ser uma nova variável de análise. Já com relação ao receptor, deve-se avaliar a possibilidade de troca do fototransistor por outro componente mais eficiente, já que esse pode também ser um fator limitante na frequência máxima de transmissão. Não limitado a esses, outros pontos de melhoria ainda devem ser analisados.

Outro ponto, pensando em relação à coleta dos mesmos dados, a fim de aumentar a base de dados, seria criar um sistema capaz de automatizar essa coleta, já que, por ora, a alteração de uma configuração de teste é completamente feita de forma manual, tanto à nível de *hardware* quanto à nível de código.

Outras sugestões de trabalhos futuros estão em aplicações que utilizam o que foi desenvolvido neste projeto. Por exemplo, sistemas de transmissão de dados de áudio, vídeo e imagem, sistemas de verificação de identidade, sistemas de iluminação junto à comunicação, entre outros.

Por fim, coloca-se também em trabalhos futuros, o estudo da razão das máximas frequências de transmissão de dados ocorrer sempre na faixa de 50% a 70% da distância máxima de transmissão.

REFERÊNCIAS

ALDERFER, Rob. Wi-Fi Spectrum: Exhaust Looms. Available at SSRN 2411645, 2013. Citado na página 21.

ARDUINO. **Introduction**. c2021. Disponível em:
<https://www.arduino.cc/en/guide/introduction>. Acesso em: 20 de fevereiro de 2021.
 Citado na página 35.

_____. **Mega 2560 Rev3**. c2022. Disponível em:
<https://docs.arduino.cc/hardware/mega-2560>. Acesso em: 12 de janeiro de 2022.
 Citado nas páginas 36, 101.

_____. **Nano**. c2022. Disponível em: <https://docs.arduino.cc/hardware/nano>.
 Acesso em: 12 de janeiro de 2022. Citado na página 39.

ATMEL CORPORATION. **ATmega328P**. California, USA, 2015. Citado na página 39.

_____. **Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V**. California, USA, 2014.
 Citado nas páginas 36–38, 40.

BRENNAND, Edna G. DE GÓES; BRENNAND, Eládio J. DE GÓES. Cognição e redes abertas: a informação interativa como coração dos sistemas inteligentes. **Ciências & Cognição**, v. 10, 2007. Citado na página 29.

COWLEY, Jhon. **Communications And Networking: An Introduction**. 1. ed.
 Staffordshire, Inglaterra: Springer, 2007. Citado nas páginas 30, 31.

ELECTRONICS WORKSHOPS. **DATA TRANSMISSION IN THE FORM OF LIGHT (LIFI COMMUNICATION) USING ARDUINO**. 2020. Disponível em:
<https://electronicsworkshops.com/2020/06/28/data-transmission-in-the-form-of-light-lifi-communication-using-arduino/>. Acesso em: 07 de fevereiro de 2022. Citado nas páginas 24, 89.

ELPROCUS. **UART Communication : Block Diagram and Its Applications**. Disponível em:
<https://www.elprocus.com/basics-of-uart-communication-block-diagram-applications/>. Acesso em: 23 de julho de 2021. Citado na página 32.

FAHAD, Engr. **Visible Light Communication “VLC” or LIFI Project using Arduino**. 2020. Disponível em: <https://www.electronicclinic.com/visible-light-communication-vlc->

[or-lifi-project-using-arduino/](#). Acesso em: 07 de fevereiro de 2022. Citado nas páginas 24, 89.

GRIDLING, Gunther; WEISS, Bettina. **Introduction to Microcontrollers**: Courses 182.064 & 182.074. [S.I.: s.n.], 2007. Citado nas páginas 34, 35.

HAAS, Harald *et al.* What is LiFi? **Journal of lightwave technology**, IEEE, v. 34, n. 6, p. 1533–1544, 2016. Citado na página 33.

HABASH, Riadh WY *et al.* Recent advances in research on radiofrequency fields and health: 2004–2007. **Journal of Toxicology and Environmental Health, Part B**, Taylor & Francis, v. 12, n. 4, p. 250–288, 2009. Citado na página 28.

HAMED, Ahed; ODEH, Alaa; AFANEH, Maram. Visible light communication (Sending data through light), 2015. Citado nas páginas 24, 89.

HELP NET SECURITY. **Number of connected devices reached 22 billion, where is the revenue?** 2019. Disponível em:
<https://www.helpnetsecurity.com/2019/05/23/connected-devices-growth/>. Acesso em: 20 de fevereiro de 2021. Citado na página 21.

HU, Pengfei *et al.* PLiFi: Hybrid WiFi-VLC networking using power lines. In: PROCEEDINGS of the 3rd Workshop on Visible Light Communication Systems. [S.I.: s.n.], 2016. P. 31–36. Citado na página 23.

KAUSHAL, Hemani; KADDOUM, Georges. Underwater optical wireless communication. **IEEE access**, IEEE, v. 4, p. 1518–1547, 2016. Citado nas páginas 22, 23.

KHAN, Latif Ullah. Visible light communication: Applications, architecture, standardization and research challenges. **Digital Communications and Networks**, Elsevier, v. 3, n. 2, p. 78–88, 2017. Citado nas páginas 21, 23, 33.

LADDHA, Neha R.; THAKARE, A. P. A Review on Serial Communication by UART. **International Journal of Advanced Research in Computer Science and Software Engineering**, v. 3, 2013. Citado na página 32.

LATHI, B. P.; ZHI DING. **Sistemas de Comunicações Analógicos e Digitais Modernos**. 4. ed. Rio de Janeiro: LTC, 2012. Citado nas páginas 29–31.

MATHEUS, Luiz *et al.* The internet of light: Impact of colors in LED-to-LED visible light communication systems. **Internet Technology Letters**, Wiley Online Library, v. 2, n. 1, e78, 2019. Citado na página 34.

MATHEUS, Luiz Eduardo Mendes *et al.* Visible light communication: concepts, applications and challenges. **IEEE Communications Surveys & Tutorials**, IEEE, v. 21, n. 4, p. 3204–3237, 2019. Citado nas páginas 21, 91.

MATHEUS, Luiz Eduardo Mendes *et al.* Livro de Minicursos SBRC 2017. In: Belém, PA: Sociedade Brasileira de Computação, 2017. Comunicação por Luz Visível: Conceitos, Aplicações e Desafios, p. 247–296. Citado nas páginas 22, 23.

QURESHI, Umair Mujtaba *et al.* RF path and absorption loss estimation for underwater wireless sensor networks in different water environments. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 16, n. 6, p. 890, 2016. Citado na página 23.

RAJAGOPAL, Sridhar; ROBERTS, Richard D; LIM, Sang-Kyu. IEEE 802.15. 7 visible light communication: modulation schemes and dimming support. **IEEE Communications Magazine**, IEEE, v. 50, n. 3, p. 72–82, 2012. Citado na página 33.

RF WIRELLES WORLD. **Advantages of Serial Interface | disadvantages of Serial Interface**. c2012. Disponível em: <https://www.rfwireless-world.com/Terminology/Advantages-and-Disadvantages-of-Serial-Interface.html>. Acesso em: 23 de julho de 2021. Citado na página 30.

SAADSAIF0333. **Data Transfer Using LED Light (Li-Fi) © MIT**. 2020. Disponível em: https://create.arduino.cc/projecthub/saadsaif0333/data-transfer-using-led-light-li-fi-c0275b?ref=similar&ref_id=363849&offset=0. Acesso em: 07 de fevereiro de 2022. Citado nas páginas 23, 89.

SPARKFUN. **TEMT6000 Ambient Light Sensor Hookup Guide**. [2003 - 2022]. Disponível em: <https://learn.sparkfun.com/tutorials/temt6000-ambient-light-sensor-hookup-guide/all>. Acesso em: 26 de janeiro de 2022. Citado na página 40.

TIPLER, Paul; MOSCA, Gene. **Física para Cientistas e Engenheiros: Eletricidade e Magnetismo, Óptica**. 6. ed. [S.l.]: LTC, 2009. v. 2. Citado na página 27.

UGWEJE, Okechukwu. Radio Frequency and Wireless Communications Security. **The University of Akron, Ohio**, 2004. Citado na página 27.

UNIVERSITY OF BERGEN. **The Electromagnetic spectrum.** 2021. Disponível em: <https://www.uib.no/en/hms-portalen/75292/electromagnetic-spectrum>. Acesso em: 20 de fevereiro de 2021. Citado na página [29](#).

VISHAY. **TEMT6000X01 PRODUCT INFORMATION.** c2022. Disponível em: <https://www.vishay.com/product?docid=81579>. Acesso em: 14 de janeiro de 2022. Citado na página [40](#).

YATEBTS. **Radio waves.** c2019. Disponível em: <https://yatebts.com/documentation/concepts/radio-waves/>. Acesso em: 20 de fevereiro de 2021. Citado na página [28](#).

Apêndices

APÊNDICE A – MATERIAIS

A.1 LISTA COMPLETA DE MATERIAIS

Os materiais utilizados ao longo do projeto foram:

- 1 x Plataforma de desenvolvimento Arduino Mega Rev3;
- 1 x Plataforma de desenvolvimento Arduino Nano;
- 2 x *Protoboards* 400 pontos;
- 1 x LED branco (3V ~ 3.3V / 20 mA);
- 1 x Resistor de 330 Ω (1/4 W);
- 2 x Resistores de 1 k Ω (1/4 W);
- 1 x Resistor de 10 k Ω (1/4 W);
- 2 x Resistores de 100 k Ω (1/4 W);
- 1 x Amplificador operacional duplo LM358N;
- 1 x Módulo TEMT6000 sensor de luz ambiente;
- 13 x Cabos de conexão *jumper*;
- 1 x Cabo USB 2.0 blindado (macho tipo A para macho tipo B) de 30 cm;
- 1 x Cabo USB 2.0 blindado (macho tipo A para macho tipo mini) de 30 cm;
- 2 x Cabos extensores USB 2.0 (fêmea tipo A para fêmea tipo A) de 2 m;
- 2 x Portas USB 3.0 conectadas a um dispositivo com sistema operacional Windows 10;
- 1 x Multímetro digital DT830-B;
- 1 x Fita métrica de 1.5 m;

Os softwares utilizados no projeto final foram:

- Arduino IDE 1.8.16;
- ATOM IDE 1.58.0, configurado para programação em Python 3.10;

Durante o desenvolvimento inicial foi utilizado um LDR, o qual foi substituído pelo Módulo TEMT6000.

Os cabos extensores USB foram utilizados apenas para uma melhor manipulação da distância entre os módulos. Para os casos avaliados, não foi observado variação da coleta com a utilização dos extensores.

O multímetro foi utilizado para validar as tensões ao longo do circuito, quando necessário, e a fita métrica para medir a distância entre os módulos.

A.2 MATERIAIS DO TRANSMISSOR

O transmissor foi desenvolvido com os seguintes materiais:

- 1 x Plataforma de desenvolvimento Arduino Nano;
- 1 x Protoboard 400 pontos;
- 1 x Resistor de 330Ω (1/4 W);
- 1 x LED branco (3V ~ 3.3V / 20 mA);
- 3 x Cabos de conexão *jumper*;
- 1 x Cabo USB 2.0 blindado (macho tipo A para macho tipo mini) de 30 cm.

A.3 MATERIAIS DO RECEPTOR

Para a implementação final do receptor, foram utilizados os seguintes componentes:

- 1 x Plataforma de desenvolvimento Arduino Mega Rev3;
- 1 x Protoboard 400 pontos;
- 2 x Resistores de $1 k\Omega$ (1/4 W);
- 1 x Resistor de $10 k\Omega$ (1/4 W);
- 2 x Resistores de $100 k\Omega$ (1/4 W);
- 1 x Módulo TEMT6000 sensor de luz ambiente;
- 1 x Amplificador operacional duplo LM358N;
- 10 x Cabos de conexão *jumper*;
- 1 x Cabo USB 2.0 blindado (macho tipo A para macho tipo B) de 30 cm.

APÊNDICE B – CÓDIGOS EXEMPLO

Os trechos de código a seguir foram compilados no ambiente de desenvolvimento Arduino IDE 1.18.6. Pelo fato da IDE já possuir um gerenciador de bibliotecas, essas serão omitidas no exemplos a seguir. Os códigos elaborados foram feitos com base na placa Arduino Mega 2560 e estão disponíveis no [Apêndice B](#). O bit correspondente, bem como o registrador, de cada pino de cada porta de entrada e saída podem ser obtidos através do mapa de pinagem do *hardware (ARDUINO, c2022a)*.

No [Código B.1](#), pode-se observar 3 diferentes formas de definir o pino 2 (D2) da plataforma como entrada, 3 diferentes formas de definir o pino 3 (D3) como saída, 3 diferentes formas de como ler os pinos 2 e 3 e por fim 3 diferentes formas de alterar o valor do pino 3. Através do mapa de pinagem, podemos ainda observar que os pinos 2 e 3 encontram-se nos registradores “E”, nos bits 4 e 5 respectivamente.

A operação lógica ‘não e’, para definir um valor, assegura que os demais *bits* daquele registrador não sofram alteração. A operação lógica ‘ou’, para definir um valor alto, assegura que os demais *bits* daquele registrador não sofram alteração. Números binários de 8 *bits* podem ser escritos tanto na forma ‘Bxxxxxxxx’, quanto na forma ‘0bxxxxxxxx’, sendo “x” valor 0 ou 1.

No [Código B.2](#), há a ativação e configuração de interrupções no sistema. Com base na [Seção 2.7.1.1.2](#), é possível compreender o código. Além disso, os comentários o deixam auto-explicativo.

Por fim, as funções *sei()* e *cli()*. Tratam-se de funções presentes na biblioteca *Arduino.h*, sendo que a primeira habilita as interrupções globais, enquanto a segunda desabilita as interrupções globais.

Registradores devem ser manipulados, quando fora de uma *ISR*, apenas com todas as interrupções desativadas, já que essas podem comprometer o fluxo desejado de valores das variáveis.

B.1 EXEMPLO 01

Código B.1 – Código exemplo 1

```

1 // 'DDRx = DDRx &' pode ser reduzido a 'DDRE &='
2 // Define pino 2 como entrada
3 DDRE = DDRE & ~(B00010000); // via 8 bits direto
4 DDRE = DDRE & ~(B1<<DDE4); // via deslocamento com base nome do
      bit no registrador
5 DDRE = DDRE & ~(B1<<4); // via deslocamento com base no indice
      desejado
6

```

```

7   {...}
8
9   // 'DDRx = DDRx |' pode ser reduzido a 'DDRE |='
10  // Define pino 3 como saída
11  DDRE = DDRE | B00100000; // via 8 bits direto
12  DDRE = DDRE | B1<<DDE5; // via deslocamento com base nome do bit
     no registrador
13  DDRE = DDRE | B1<<5; // via deslocamento com base no índice
     desejado
14
15 {...}
16
17 // Leitura do valor no pino 2
18 int pino2valor = PINE & B00010000; // via 8 bits direto
19 int pino2valor = PINE & B1<<PINE4; // via deslocamento com base
     nome do bit no registrador
20 int pino2valor = PINE & B1<<4; // via deslocamento com base no
     índice desejado
21
22 {...}
23
24 // Leitura do valor no pino 3
25 int pino3valor = PINE & B00100000; // via 8 bits direto
26 int pino3valor = PINE & B1<<PINE5; // via deslocamento com base
     nome do bit no registrador
27 int pino3valor = PINE & B1<<5; // via deslocamento com base no
     índice desejado
28 {...}
29
30 // 'PORTx = PORTx &' pode ser reduzido a 'PORTx &='
31 // 'PORTx = PORTx |' pode ser reduzido a 'PORTx |='
32 // via 8 bits direto
33 PORTE = PORTE | B00100000; // Escrita de valor alto no pino 3
34 PORTE = PORTE & ~(B00100000); // Escrita de valor baixo no pino 3
35 // via deslocamento com base nome do bit no registrador
36 PORTE = PORTE | B1<<PE5; // Escrita de valor alto no pino 3
37 PORTE = PORTE & ~(B1<<PE5); // Escrita de valor baixo no pino 3
38 // via deslocamento com base no índice desejado
39 PORTE = PORTE | B1<<5; // Escrita de valor alto no pino 3
40 PORTE = PORTE & ~(B1<<5); // Escrita de valor baixo no pino 3

```

B.2 EXEMPLO 02

Código B.2 – Código exemplo 2

```

1 // Desabilita interrupção global
2 cli();
3
4 // ##### Configura TIMER 1 #####
5 // Inicializa TIMER1
6 TCCR1A = 0;
7 TCCR1B = 0;
8 // Valor inicial do contador do TIMER1
9 TCNT1 = 0;
10
11 // Configura TIMER 1 para comparação com CTC

```

```

12 // Modo CTC faz comparacao direta com contagem ascendente
13 // [0->65536]
14 TCCR1A = B00000000; // [7:6] -> Comparacao no Canal A:
15 // NORMAL [00];
16 // [5:4] -> Comparacao no Canal B: NORMAL
17 // [00];
18 // [3:2] -> RESERVADOS;
19 // [1:0] -> Waveform Generation Mode:
20 // 00 pois com TCCR1B [xxxx1xxx]
21 // forma CTC (CLEAR TIME ON COMPARE)
22 // reset o contador TCNT1 quando
23 // a flag de comparacao
24 // e' acionada.
25
26
27 // Interrupcao por timer sera habilitada durante a interrupcao
28 // externa (FALLING EDGE)
29 TIMSK1 = B00000010; // [7:6] -> RESERVADOS;
30 // [5] -> Habilita interrupcao por INPUT
31 // ;
32 // [4:3] -> RESERVADOS;
33 // [2] -> Habilita interrupcao por
34 // COMPARACAO no Canal B;
35 // [1] -> Habilita interrupcao por
36 // COMPARACAO no Canal A;
37 // [0] -> Habilita interrupcao por
38 // OVERFLOW do CONTADOR1;
39
40 // Valor a ser comparado no Canal A
41 OCR1A = 30000;
42
43 //##### Configura interrupcao externa (LOW, Any, Falling Edge ,
44 // Rising Edge) #####
45 // [00] LOW LEVELS
46 // [01] ANY EDGE
47 // [10] FALLING EDGE
48 // [11] RISING EDGE
49 EICRA = B00000000; // [7:6] INT3 - PIN18
50 // [5:4] INT2 - PIN19
51 // [3:2] INT1 - PIN20
52 // [1:0] INTO - PIN21
53 EICRB = B00000010; // [7:6] INT7 - NA
54 // [5:4] INT6 - NA
55 // [3:2] INT5 - PIN3
56 // [1:0] INT4 - PIN2
57
58 // Ativa Ext Interrupt para INT7~0
59 EIMSK = B00010000; (INT4)
60
61 // Habilita interrupcao global
62 sei();

```


APÊNDICE C – STRINGS

C.1 STRING DE 100 CARACTERES (100 BYTES)

UaSGzuVmckQSbaSWIvVlvpkymWEzNtj8HwH61HYYCFKKRm2eMr6WoscaAWYzqVa0IDSw0b6DzqLbKO9E4IKPbkY8xZHlfLgfsF8e

C.2 STRING DE 500 CARACTERES (500 BYTES)

jRAdmzYPhjhYE1RGURuDgMPBNuChB40209sHtyTPwE5IBLu1ppLjFdNNG6BVkh05463G8fuUKmfYVP07N8qDJ1wn9mHFnX60CQIunQbLtDUBKI5ovpy8xPzmaa7C4roKiD5xt8NSAYv4ERqu58R316TE3qzrMhimqGu2WkUmjqvJlpqOGsJNesr0JjioYn om1I6WWhClia5fGQUeJGT50JpFktpYx2GMYoYqrzYaYfa75wkqtMwdt5K5xPsuKsx up06LlcbfCfZXcm9PB5FdkL67XFTg3kRJqPbtYxxS2qMBulx3hZllqpJuo2aUc2dRgHR qAHsJZJd00ADmxJJZ0cehE7fyIPtdolQJFxKWDe06trKMHEyrfVX8N8nPQWKeZna mH1t6yCCo7S1MTBBPctjhwtGnFAHTvOf60gYIkbyHY5Eu2VYThJHI5k5BWV1TyKT UyXO2tjTxSfyd6FsyFxMNqpT1Z3VL2I1OdvnCZAhEAEZ4BZ

C.3 STRING DE 1000 CARACTERES (1000 BYTES)

eAvoCErOLrGuQK5QEfHdgHDaCCeO6suIZZkfY4XCA34QfZfgfJPRZJGgyBZ qAjpwpxdnvr7vNgIr1JaSxOVS5zySV5dLHj1ewCT9iwKQgJsr6hJwJBecgSaTHMFNLk 9XII4qlel3yP4eRWYGvk2nN4JAihbXA77pjEUJzZI7xwNx5ikm43JJmVkkfHZkIV5adXI vHI42gESPsf2ABa2TsEZtfKqwnVDf8qLidzRdLq2AodM7mhpu3tAxXQhlWnRtvqwbF1 D2twVMI8uNgBqw54ttMbO5ObFg3vjsUKVhD62InCQ194xs9t1ZAuKSjdNfWrVxIqu aLEWNrEpD2Y95dgiWtS6RzLocX8nR8TQQhZ2YdoJJ6hVa0u0qlvXyNhV8IGQbday4 uOAA2qRCBLuyVj7RzcflyzARd2PDT8zqHbX0DBqaNrPlqO9UsNSImf4cCGKrAeXA5 2EgWr9ekADAuJILJnm2EQnm6vvOps0NrBBAIGfQLLMQuDC2SBF1ocgXZbzJCutE FX7UVmVe2FePs6eQnTopIgj6ieqQnkgnuhvAahUfAiEtDvyL4BuSgDEPcBGp1VAsAu jaQH4AFIOYMx7ephKCQokZZhcPnvAXdphAGArgw82XNAZbJAtAvHQinwsGEXsQAj x2KvUJmJa0kesyHSLvirrc3C1sMnNKLNCOyi3I4X0AGZOZNUY2RoLsZPrfsx8vO5R iW0UzzJ285W2gzNJR1hMmxzfd4uZfj7IEQeghA3MhlaG3Fc12v86baArI4SSqv9gWa wobkHRav0bCaqdHw6UcTNW0b7g5TWpE1ykfaSYB1W6YvTNnoaoEPPsN5tOqOft9 GcGvIKyX41JBmvQKnr7Kk31wRx545Dwwa20EeGtXGlfr2ua2ivBForDvDaKyINfVthk BTcCc6KJHKFdgPcXkWQjHtyY2PzDX485Tq7qS5WM6FEgikt9oFbWawTo3E2EIQF UsnQ6ECkRFikt9oFbWawTo3E

APÊNDICE D – CÓDIGO DO TRANSMISSOR

D.1 PARTE 01

Código D.1 – Código do transmissor - Parte 01

```

1 // NANO PORT COM 4 (USB ESQ)
2 -----
3 // Port direct pin 2 -> PORTD[2] 00000$00
4 // !!! ARDUINO IDE 1.18.6 !!!
5 // May fail on loading to board in newer IDE versions
6 // Older IDE versions not tested
7
8 // VERSAO COM INTERRUPCAO DO TIMER1 POR COMPARACAO
9
10
11#define START_STATE 0 // Estado de inicializacao , tempo para iniciar o
12// receptor
12#define START_BIT_STATE 1 // Estado de envio do Start Bit
13#define DATA_BITS_STATE 2 // Estado de envio dos Bits do char
14#define STOP_BIT_STATE 3 // Estado de envio do Stop Bit
15
16#define STRING_SIZE 100
17
18#define FREQUENCY 500
19#define OCR1A_FROM_FREQUENCY 16000000/FREQUENCY // NÃO MEXER
20
21char string[STRING_SIZE+1] = ""; // SUBSTITUIR "" PELA STRING
22// DESEJADA
22
23int stringLengthSent; //Qtd de char enviados
24
25int state; // Estado atual do sistema
26boolean stateChangeLocker; // Habilita execução do código de um estado
27
28int dataByteIndex; // indice do bit no byte de dado
29
30long systemStartCounter = 0; // Valor alto entre envio de frases
31// completas;
31// É necessário para que o buffer do
32// receptor seja capaz de
32// "empurrar" valor contido para fora e
32// tenha o índice resetado
33
34// Deve ser uma valor maior que OCR1A e
34// suficiente para o buffer,
35// do receptor printar no console Serial ,
35// pois assim o período em Idle
36// do sistema fará um auto ajuste (após
36// algumas transmissões) na saída
37// da frase no buffer do receptor caso o
37// sistema seja iniciado no meio
38// do envio de uma transmissão de dados.
39
40int finishedTransmissions; // Quantidade de transmissões realizadas

```

D.2 PARTE 02

Código D.2 – Código do transmissor - Parte 02 basicstyle

```

1 void setup() {
2     Serial.begin(115200);      // Habilita o Serial
3
4     pinMode(LED_BUILTIN, OUTPUT);    // Desliga o LED integrado
5     digitalWrite(LED_BUILTIN, LOW);
6
7     DDRD = DDRD | B00000100; // inputs e outputs; pin 2 Output
8     PORTD = B00000100; // valor alto ou baixo; pin 2 Alto
9
10    stringLengthSent = 0;
11
12    state = START_STATE;
13    stateChangeLocker = 1;
14    dataByteIndex = 0;
15
16    // Intervalo entre transmissoes em segundos
17    systemStartCounter = (long) FREQUENCY * 5; // COLOCAR FREQUENCY * y
18        , para y segundos
19
20    // Desabilita interrupção global
21    cli();
22
23    // Inicializa TIMER1
24    TCCR1A = 0;
25    TCCR1B = 0;
26    // Valor inicial do contador do TIMER1
27    TCNT1 = 0;
28
29    // Configura TIMER 1 para comparacao com CTC
30    // Modo CTC faz comparação direta com contagem ascendente
31        [0->65536]
32    TCCR1A = B00000000;      // [7:6] -> Comparacao no Canal A: NORMAL
33        [00];
34
35        // [1:0] -> Waveform Generation Mode:
36        //          00 pois com TCCR1B [xxxx1xxx]
37        //          forma CTC (CLEAR TIME ON COMPARE)
38        //          reset o contador TCNT1 quando
39        //          a flag de comparacao
40        //          é acionada.
41
42    TCCR1B = B00001001;      // [4:3] -> Waveform Generation Mode;
43        // [2:0] -> Prescaler (setado em 1).
44
45    TIMSK1 = 0b00000010;    // [1]    -> Habilita interrupcao por
46        // COMPARACAO no Canal A;
47
48    // Valor a ser comparado no Canal A
49    OCR1A = OCR1A_FROM_FREQUENCY; //usar valor par preferencialmente
50
51    // Habilita interrupção global
52    sei();
53}

```

D.3 PARTE 03

Código D.3 – Código do transmissor - Parte 03

```
1void loop() {
2    if(stateChangeLocker==1){
3
4        // Matém o LED aceso por 5s.
5        if(state == START_STATE && stateChangeLocker==1) {
6            PORTD = B00000100;
7            stateChangeLocker=0;
8        }
9
10       // Faz o envio do Start Bit (nível baixo).
11       if(state == START_BIT_STATE && stateChangeLocker==1) {
12           PORTD = B00000000;
13           stateChangeLocker=0;
14           if(stringLengthSent == 0) {
15               Serial.println("SENDING... ");
16           }
17       }
18
19       // Faz o envio dos Bits do Char;
20       // Ao enviar 8 bits (baudDataBitIndex==8), vai para o Stop Bit
21       // e incrementa para o próximo caracter da string (
22       // stringLengthSent++).
23       if(state == DATA_BITS_STATE && stateChangeLocker==1) {
24           // Bits de dados (LSB para MSB)
25           if((string[stringLengthSent] & (0x01 << dataByteIndex)) != 0) {
26               PORTD = B00000100;
27           } else {
28               PORTD = B00000000;
29           }
30           dataByteIndex++;
31
32           if(dataByteIndex >= 8) {
33               stringLengthSent++;
34           }
35           stateChangeLocker=0;
36       }
37
38       // Faz o envio do Stop Bit (nível alto) e volta para o Start
39       // Bit;
40       // Se a qtd de char enviados for maior ou igual ao tamanho da
41       // frase
42       // Volta para estado de inicialização do sistema.
43       if(state == STOP_BIT_STATE && stateChangeLocker==1) {
44           PORTD = B00000100;
45           //if(stringLengthSent == STRING_SIZE) {
46           //    Serial.println("SENT ");
47           //}
48           stateChangeLocker=0;
49       }
50   }
51 }
```

D.4 PARTE 04

Código D.4 – Código do transmissor - Parte 04

```
1 // Interrupção por COMPARACAO do TIMER/CONTADOR1 (CANAL A)
2 ISR(TIMER1_COMPA_vect) {
3     if(state == START_STATE) {
4         systemStartCounter--;
5         if(systemStartCounter <= 0L) {
6             state=START_BIT_STATE;
7             stateChangeLocker = 1;
8         }
9     } else if(state == START_BIT_STATE) {
10        state=DATA_BITS_STATE;
11        stateChangeLocker = 1;
12    } else if(state == DATA_BITS_STATE) {
13        if(dataByteIndex >= 8) {
14            dataByteIndex = 0;
15            state=STOP_BIT_STATE;
16        }
17        stateChangeLocker = 1;
18    } else if(state == STOP_BIT_STATE) {
19        if(stringLengthSent >= STRING_SIZE) {
20            stringLengthSent = 0;
21            state = START_STATE;
22            // DELAY ENTRE TRANSMISSÕES EM SEGUNDOS
23            // USER FREQUENCY / 2 para 0.5 Segundos
24            systemStartCounter = (long) FREQUENCY * 0.5;
25        } else {
26            state=START_BIT_STATE;
27        }
28        stateChangeLocker = 1;
29    }
30}
```

APÊNDICE E – CÓDIGO DO RECEPTOR

E.1 PARTE 01

Código E.1 – Código do receptor - Parte 01

```

1 // MEGA PORT COM 3 or COM 5 (USB DIR)
2 //-----
3 // Port direct pin 2 -> PORTE[4] 000$0000
4 // !!! ARDUINO IDE 1.18.6 !!!
5 // May fail on loading to board in newer IDE versions
6 // Older IDE versions not tested
7
8 // VERSAO COM INTERRUPCAO DO TIMER1 POR COMPARACAO E
9 // COM INTERRUPCAO POR BORDA DE DESCIDA EM PIN 2
10
11#define IDLE_STATE 0 // Estado de aguardo do proximo Start Bit
12#define START_BIT_STATE 1 // Estado de envio do Start Bit
13#define DATA_BITS_STATE 2 // Estado de envio dos Bits do char
14#define STOP_BIT_STATE 3 // Estado de envio do Stop Bit
15
16#define STRING_SIZE 100
17
18#define FREQUENCY 500
19#define OCR1A_FROM_FREQUENCY 16000000/FREQUENCY // NÃO MEXER
20#define OCR1A_FROM_FREQUENCY_HALF OCR1A_FROM_FREQUENCY/2 // NÃO
               MEXER
21
22#define TRANSMISSION_LIMIT 10 //Quantidade máxima de transmissões
23
24// Delay inicial para pegar o meio do bit (ponto ótimo)
25boolean initialdelay;
26
27// Aloca 1300 espaços para char (buffer de recebimento) para evitar
   multiplos Serial.print byte a byte
28// Valor máximo devido a memória do emissor
29char *stringReceivedBuffer = (char*) calloc(1301, sizeof(char));
30
31char charBits; // 0 char ASCII de fato (8 bits -> 1 byte)
32int bitReceived; // Valor do bit atual
33int bytesReceivedAmount; // Quantidade de bytes recebidos até o
                           momento
34
35int state; // Estado atual do sistema
36boolean stateChangeLocker = 0; // Habilita execução do código de um
                                estado
37
38int dataByteIndex; // indice do bit no byte de dado
39
40int finishedTransmissions; // Quantidade de transmissões realizadas

```

E.2 PARTE 02

Código E.2 – Código do receptor - Parte 02

```

1 void setup() {
2     Serial.begin(115200);      // Habilita o Serial
3
4     pinMode(LED_BUILTIN, OUTPUT);    // Desliga o LED integrado
5     digitalWrite(LED_BUILTIN, LOW);
6
7     stringReceivedBuffer[STRING_SIZE] = '\0'; // Fim de string
8
9     DDRE = DDRE & ~(B00010000); // pin 2 DIGITAL D2 (PORTA E4) input
10
11    charBits = 0;
12    bitReceived = 0;
13    bytesReceivedAmount = 0;
14
15    state = IDLE_STATE;
16
17    stateChangeLocker = 0;
18    dataByteIndex = 0;
19    finishedTransmissions = 0;
20
21    // Desabilita interrupção global
22    cli();
23
24    // ##### Configura TIMER 1 #####
25    TCCR1A = 0;
26    TCCR1B = 0;
27
28    TCNT1 = 0; // Valor inicial do contador do TIMER1
29
30    TCCR1A = B00000000;
31    TCCR1B = B00001001;
32
33    // Interrupção por timer será habilitada durante a interrupção
34    // externa (FALLING EDGE)
35    TIMSK1 = B00000010; // [1] -> Habilita interrupção por
36    // COMPARAÇÃO no Canal A;
37
38    // Valor a ser comparado no Canal A
39    OCR1A = OCR1A_FROM_FREQUENCY; //usar valor par de preferência
40
41    //##### Configura interrupção externa #####
42    // [00] LOW LEVELS [01] ANY EDGE [10] FALLING EDGE RISING EDGE
43    EICRA = B00000000; // Interrupções não utilizadas
44    EICRB = B00000010; // [1:0] INT4 - PIN2 (D2) - Sendo utilizado no
45    // momento
46    // Ativa Ext Interrupt para INT7~0
47    EIMSK = B00010000;
48
49    // Habilita interrupção global
50    sei();
51
52}

```

E.3 PARTE 03

Código E.3 – Código do receptor - Parte 03

```
1 void loop() {
```

```
2     if(stateChangeLocker == 1){
3
4         if(state == IDLE_STATE) {
5             stateChangeLocker = 0;
6         }
7
8         if(state == START_BIT_STATE) {
9             // Desativa interrupções externas
10            cli(); EIMSK = B00000000; sei();
11
12            stateChangeLocker=0;
13        }
14
15         if(state == DATA_BITS_STATE) {
16             bitReceived = PINE>>4 & B1;
17             charBits = charBits | (bitReceived << dataByteIndex);
18             dataByteIndex++;
19             stateChangeLocker=0;
20         }
21
22         if(state == STOP_BIT_STATE) {
23
24             if(PINE>>4 & B1 == 0) { // Não é um StopBit
25                 stringReceivedBuffer[bytesReceivedAmount] = '?';
26             } else if ((charBits > 47 && charBits < 58) ||
27                         (charBits > 64 && charBits < 91) ||
28                         (charBits > 96 && charBits < 123)) { // Temos
29                 um alfanumerico
30                 stringReceivedBuffer[bytesReceivedAmount] = charBits;
31             } else { // Temos um falso positivo StopBit
32                 stringReceivedBuffer[bytesReceivedAmount] = '?';
33             }
34             bytesReceivedAmount++;
35             if(bytesReceivedAmount >= STRING_SIZE){
36                 bytesReceivedAmount = 0;
37                 Serial.println(stringReceivedBuffer);
38                 finishedTransmissions++;
39             }
40
41             charBits = 0;
42             stateChangeLocker=0;
43
44             cli(); EIMSK = B00010000; sei();
45
46             state = IDLE_STATE;
47         }
48
49         if(finishedTransmissions == TRANSMISSION_LIMIT) {
50             Serial.print("FINISHED"); delay(1000); exit(0);
51         }
52     }
```

E.4 PARTE 04

```
1 // Interrupção externa (FALLING EDGE) em pin 2
2 ISR(INT4_vect) {
3
4     state = START_BIT_STATE;
5
6     // Metade do tempo definido em OCR1A - delay inicial ponto
7     amostragem otimo
8     TCNT1 = OCR1A_FROM_FREQUENCY_HALF;
9
10    // Com base, em TCNT1, para amostragem no ponto médio do bit
11    initialdelay = 1;
12
13    stateChangeLocker = 0;
14}
15 // Interrupção por COMPARACAO do TIMER/CONTADOR1 (CANAL A)
16 ISR(TIMER1_COMPA_vect) {
17     if(state == IDLE_STATE) {
18         stateChangeLocker = 1;
19     } else if(state == START_BIT_STATE) {
20         stateChangeLocker = 1;
21         if(initialdelay == 1) {
22             initialdelay = 0;
23         } else if (initialdelay == 0) {
24             state = DATA_BITS_STATE;
25         }
26     } else if(state == DATA_BITS_STATE) {
27         stateChangeLocker = 1;
28         if(dataByteIndex >= 8) {
29             dataByteIndex = 0;
30             state = STOP_BIT_STATE;
31         }
32     }
33 }
```

APÊNDICE F – CÓDIGO DE COMPILAÇÃO E BUSCA DOS DADOS GERADOS

F.1 CÓDIGO FONTE COMPLETO

Código F.1 – Código para compilação e filtragem dos dados coletados

```

1 import os
2 import json
3
4
5 rootDirectory = 'C:/Users/Daniel/Documents/VLC_DADOS_CLEAN'
6 # set base strings
7 with open(f'{rootDirectory}/100BytesString.txt') as string:
8     String100Bytes = string.read()
9 with open(f'{rootDirectory}/500BytesString.txt') as string:
10    String500Bytes = string.read()
11 with open(f'{rootDirectory}/1000BytesString.txt') as string:
12     String1000Bytes = string.read()
13
14
15 # Attemp to make os.walk sort numbers
16 def renameToMilimeters():
17     for currentPath, subdirs, files in sorted(os.walk(rootDirectory)):
18         for subdir in subdirs:
19             if(subdir.split('.')[ -1] == '0cm'):
20                 newSubdirName = (f'{subdir.split(".")[0]}0mm')
21                 os.rename(f'{currentPath}/{subdir}', f'{currentPath}/{newSubdirName}')
22             if(subdir.split('.')[ -1] == '5cm'):
23                 newSubdirName = (f'{subdir.split(".")[0]}5mm')
24                 os.rename(f'{currentPath}/{subdir}', f'{currentPath}/{newSubdirName}')
25
26
27 # Attemp to make os.walk sort numbers
28 def renameToCentimeters():
29     for currentPath, subdirs, files in sorted(os.walk(rootDirectory)):
30         for subdir in subdirs:
31             if(subdir[-3:] == '0mm'):
32                 newSubdirName = (f'{subdir[:len(subdir)-3]}.0cm')
33                 print(newSubdirName)
34                 os.rename(f'{currentPath}/{subdir}', f'{currentPath}/{newSubdirName}')
35             if(subdir[-3:] == '5mm'):
36                 newSubdirName = (f'{subdir[:len(subdir)-3]}.5cm')
37                 print(newSubdirName)
38                 os.rename(f'{currentPath}/{subdir}', f'{currentPath}/{newSubdirName}')
39
40
41 def adjustBytesName():
42     for currentPath, subdirs, files in sorted(os.walk(rootDirectory)):
43         for subdir in subdirs:
44             if(len(subdir) == 4 and subdir[-1:] == 'B'):
45                 print(subdir)
46                 newSubdirName = (f'0{subdir}')
47                 print(newSubdirName)
48                 os.rename(f'{currentPath}/{subdir}', f'{currentPath}/{newSubdirName}')

```

```

        newSubdirName}')
```

49

50

51 def adjustCentimetersName():

52 for currentPath, subdirs, files in sorted(os.walk(rootDirectory)):

53 for subdir in subdirs:

54 if(len(subdir) == 5 and subdir[-2:] == 'cm'):

55 print(subdir)

56 newSubdirName = (f'0{subdir}')

57 print(newSubdirName)

58 os.rename(f'{currentPath}/{subdir}', f'{currentPath}/{newSubdirName}')

59

60

61 def adjustFrequencyName():

62 for currentPath, subdirs, files in sorted(os.walk(rootDirectory)):

63 for file in files:

64 if(len(file) == 9 and file[-6:] == 'Hz.txt'):

65 print(file)

66 newFileName = (f'0{file}')

67 print(newFileName)

68 os.rename(f'{currentPath}/{file}', f'{currentPath}/{newFileName}')

69

70

71 # Generates json for each leave folder, the only that contains files (

frequency files) with the trnamission received

72# A 'result.json' will be created at rootDirectory/xxxkxxx/xxxxB/xx.

xxcm/

73 def generateJsons():

74 print('Generating json files')

75 filesTotal = 0

76 dirswithJson = 0

77 for currentPath, subdirs, files in sorted(os.walk(rootDirectory)):

78 #print(currentPath) -- DEBUG

79 #print(subdirs) -- DEBUG

80 #print('\n') -- DEBUG

81 transmissionsLineJsonArray = []

82 for filename in files:

83 if not (filename == 'result.json' or filename == '100

BytesString.txt' or filename == '500BytesString.txt' or

84 filename == '1000BytesString.txt'):

85 filesTotal += 1

86 with open(f'{currentPath}/{filename}') as file:

87 transmissionsLines = file.readlines()

88 frequency = f'{filename}' # arquivo atual

89 distance = f'{os.path.basename(currentPath)}' #

dir acima Dist

90 bytes = f'{os.path.basename(os.path.dirname(

currentPath))}' # dir acima acima Bytes

91 voltageDivider = f'{os.path.basename(os.path.

dirname(os.path.dirname(currentPath)))}' #

dir acima acima VoltageDivider

92 cleanTransmissions = 0 # 0 erros

93 dirtyTransmissions = 0 # 0 < errorRate =<

0.050

94 failedTransmissions = 0 # 0.050 < errorRate

95

96 # For each trnamission, we're going to generate

one json object

```

97         # All json wil be put in a json array
98         if(transmissionsLines and len(transmissionsLines)
99             >= 10):
100             transmissionsLineDict = {}
101             transmissionsLineDict.update({'voltageDivider':
102                 f'{voltageDivider}'})
103             transmissionsLineDict.update({'bytes': f'{bytes}'})
104             transmissionsLineDict.update({'distance': f'{distance}'})
105             transmissionsLineDict.update({'frequency': f'{frequency[:-4]}'})
106             for index in range(0, 10):
107                 if ((f'{os.path.basename(os.path.dirname(
108                     currentPath))}') == '0100B'):
109                     if(transmissionsLines[index][16:-1] ==
110                         String100Bytes):
111                         transmissionsLineDict.update({f'{
112                             transmissionNumber{index+1}
113                             Error': '0/100'}})
114                         transmissionsLineDict.update({f'{
115                             transmissionNumber{index+1}
116                             ErrorRatio': '0'}})
117                         cleanTransmissions += 1
118             else:
119                 error = 0
120                 errorRate = .0
121                 for strIndex in range(0, 100):
122                     if not(transmissionsLines[
123                         index][16:-1][strIndex] ==
124                         String100Bytes[strIndex]):
125                         error += 1
126                 errorRate = round(error/100, 3)
127                 transmissionsLineDict.update({f'{
128                     transmissionNumber{index+1}
129                     ErrorRatio': error}})
130                 transmissionsLineDict.update(
131                     {f'transmissionNumber{index+1}'
132                         ErrorRatio': errorRate})
133             )
134             if(errorRate <= 0.050):
135                 dirtyTransmissions += 1
136             else:
137                 failedTransmissions += 1
138             elif ((f'{os.path.basename(os.path.dirname(
139                     currentPath))}') == '0500B'):
140                 if(transmissionsLines[index][16:-1] ==
141                     String500Bytes):
142                     transmissionsLineDict.update({f'{
143                         transmissionNumber{index+1}
144                         Error': '0/500'}})
145                     transmissionsLineDict.update({f'{
146                         transmissionNumber{index+1}
147                         ErrorRatio': '0'}})
148                     cleanTransmissions += 1
149             else:
150                 error = 0
151                 errorRate = .0
152                 for strIndex in range(0, 500):
153                     if not(transmissionsLines[
```

```

135                                         index][16:-1][strIndex] ==
136                                         String500Bytes[strIndex]):
137                                         error += 1
138                                         errorRate = round(error/500, 3)
139                                         transmissionsLineDict.update({f'
140                                         transmissionNumber{index+1}
141                                         ErrorRatio': error})
142                                         transmissionsLineDict.update(
143                                         {f'transmissionNumber{index+1}'
144                                         ErrorRatio': errorRate}
145                                         )
146                                         if(errorRate <= 0.050):
147                                             dirtyTransmissions += 1
148                                         else:
149                                             failedTransmissions += 1
150                                         elif ((f'{os.path.basename(os.path.dirname
151                                         (currentPath))}') == '1000B'):
152                                             if(transmissionsLines[index][16:-1] ==
153                                         String1000Bytes):
154                                             transmissionsLineDict.update({f'
155                                         transmissionNumber{index+1}
156                                         Error': '0/1000'})
157                                         transmissionsLineDict.update({f'
158                                         transmissionNumber{index+1}
159                                         ErrorRatio': '0'})
160                                         cleanTransmissions += 1
161                                         else:
162                                             error = 0
163                                             errorRate = .0
164                                             for strIndex in range(0, 1000):
165                                                 if not(transmissionsLines[
166                                         index][16:-1][strIndex] ==
167                                         String1000Bytes[strIndex]):
168                                                 error += 1
169                                             errorRate = round(error/1000, 3)
170                                             transmissionsLineDict.update({f'
171                                         transmissionNumber{index+1}
172                                         Error': error})
173                                             transmissionsLineDict.update(
174                                         {f'transmissionNumber{index+1}'
175                                         ErrorRatio': errorRate}
176                                         )
177                                         if(errorRate <= 0.050):
178                                             dirtyTransmissions += 1
179                                         else:
180                                             failedTransmissions += 1
181                                         transmissionsLineDict.update({
182                                         'cleanTransmissions': f'{cleanTransmissions}
183                                         '})
184                                         transmissionsLineDict.update({
185                                         'dirtyTransmissions': f'{dirtyTransmissions}
186                                         '})
187                                         transmissionsLineDict.update({
188                                         'failedTransmissions': f'{{
189                                         failedTransmissions}}}')
190                                         if(cleanTransmissions+dirtyTransmissions < 10)
191                                         :
192                                             cleanDirtyTransmissions = f'0{
193                                             cleanTransmissions+dirtyTransmissions}'
194                                         else:

```

```

171             cleanDirtyTransmissions =
172                 cleanTransmissions+dirtyTransmissions
173             transmissionsLineDict.update(
174                 {'overallTransmissions': f'{cleanDirtyTransmissions}/10'})
175             #firstFailedFrequencyTrnmission = False
176             if (cleanTransmissions == 10):
177                 transmissionsLineDict.update({'resultTransmission': 'SUCCESS'})
178             elif (cleanTransmissions+dirtyTransmissions >=
179                   9):
180                 transmissionsLineDict.update({'resultTransmission': 'ACCEPTABLE'})
181             else:
182                 transmissionsLineDict.update({'resultTransmission': 'FAILED'})
183             #firstFailedFrequencyTransmission = True
184
185             transmissionsLineJson = json.dumps(
186                 transmissionsLineDict, indent=4)
187             transmissionsLineJsonArray.append(
188                 transmissionsLineJson)
189             #print(transmissionsLineJsonArray) -- DEBUG
190
191             # If 2 or more trasmission from 10 failed,
192             # ignore the following frequencies for that
193             # distance
194             # if(firstFailedFrequencyTransmission):
195             #     break;
196             else:
197                 print("Less than 10 transmissions")
198                 print(f'{currentPath}/{filename}')
199                 # exit(0) -- DEBUG
200                 file.close()
201             # Since transmissionsLineJsonArray will have data only inside
202             # folders with files. Since we have files only in
203             # the lowest folders in the tree (leaves), we can ensure that
204             # condition
205             if(transmissionsLineJsonArray):
206                 # Save json data along Hz files
207                 file = open(f'{currentPath}/result.json', "w")
208                 file.write('[\n')
209                 for jsonData in transmissionsLineJsonArray[:-1]:
210                     file.write(jsonData)
211                     file.write(",\n")
212                 file.write(transmissionsLineJsonArray[-1])
213                 file.write('\n]')
214                 file.close()
215                 dirswithJson += 1
216                 #print(transmissionsLineJsonArray) -- DEBUG
217                 #print(currentPath) -- DEBUG
218             # If transmissionsLineJsonArray is empty, we're in a directory
219             # with no raw data. If there is any result.json,
220             # it SHOULD NOT be there
221             else:
222                 try:
223                     os.remove(f'{currentPath}/result.json')
224                 except OSError:
225                     pass
226
227

```

```
218     print('\nAll json files created')
219     # Files quantity verified on Windows Files Explorer
220     print(f'Scanned Files {filesTotal}')
221
222
223# Get all rootDirectory/xxxkkk/xxxxB/xx.xxcm/ and compiles in the file
224#   rootDirectory/result.json
225def compileJsons():
226    print('\nCompiling jsons to a json file in root directory')
227    transmissionsLineJsonArray = []
228    dirswithJson = 0
229    for currentPath, subdirs, files in sorted(os.walk(rootDirectory)):
230        #print(currentPath) -- DEBUG
231        #print(subdirs) -- DEBUG
232        #print('\n') -- DEBUG
233
234        for filename in files:
235            if (filename == 'result.json' and not currentPath ==
236                rootDirectory):
237                #print(currentPath) -- DEBUG
238                with open(f'{currentPath}/result.json') as resultJson:
239                    dict = json.loads(resultJson.read())
240                    for data in dict:
241                        transmissionsLineJson = json.dumps(data,
242                            indent=4)
243                        transmissionsLineJsonArray.append(
244                            transmissionsLineJson)
245                        #print(data) -- DEBUG
246                    resultJson.close()
247                    #dirswithJson+=1 --DEBUG
248
249
250# print(transmissionsLineJsonArray) -- DEBUG
251file = open('C:/Users/Daniel/Documents/VLC_DADOS_CLEAN/result.json',
252            'w')
253file.write('[\n')
254for jsonData in transmissionsLineJsonArray[:-1]:
255    file.write(jsonData)
256    file.write(",\n")
257file.write(transmissionsLineJsonArray[-1])
258file.write('\n]')
259file.close()
260# print(dirswithJson) --DEBUG
261print('\nAll json files compiled to root directory')
262
263
264# Open root Json for analysis
265def openJson():
266    with open(f'{rootDirectory}/result.json') as resultJson:
267        #data = json.loads(line) for line in resultJson
268        dict = json.loads(resultJson.read())
269        print(len(dict))
270        voltageDivider = '50k01k'
271        bytes = '0500B'
272        #distance = '04.5cm'
273        #print(f'{voltageDivider} > {bytes} > {distance}')
274        actualDistance = ''
275        previousDistance = ''
276        amount = 0
277        for data in dict:
278            actualDistance = data["distance"]
```

```

273         if(data["voltageDivider"] == voltageDivider
274             and data["bytes"] == bytes):
275                 if(actualDistance != previousDistance):
276                     print('')
277                     print(data['distance'])
278                     outputResultSearch = ''
279                     outputResultSearch += f'{data["frequency"]}'
280                     if (len(data["cleanTransmissions"]) == 1):
281                         outputResultSearch += f' 0{data["'
282                             cleanTransmissions"]}'''
283                     else:
284                         outputResultSearch += f' {data["cleanTransmissions"
285                             ""]}''
286                     if (len(data["dirtyTransmissions"]) == 1):
287                         outputResultSearch += f' 0{data[""
288                             dirtyTransmissions"]}'''
289                     else:
290                         outputResultSearch += f' {data["dirtyTransmissions"
291                             ""]}''
292                     if (len(data["failedTransmissions"]) == 1):
293                         outputResultSearch += f' 0{data[""
294                             failedTransmissions"]}'''
295                     else:
296                         outputResultSearch += f' {data[""
297                             failedTransmissions"]}'''
298                     outputResultSearch += f' {data["overallTransmissions"
299                             ""]}''
300                     outputResultSearch += f' {data["resultTransmission"]}''
301                     print(outputResultSearch)
302                     amount = amount + 1
303                     print(amount)
304                     previousDistance = data["distance"]
305                     resultJson.close()
306
307
308
309
310
311
312
313 if (__name__ == '__main__'):
314     # !!!! RENAME AND ADJUST FILENAMES TO GET SORT CORRECTED - CAREFUL
315     # !!!
316     # DO NOT NEED TO BE EXECUTED. MADE JUST FOR EASY READING WHEN
317     # FILTER JSONS
318     #renameToMilimeters()
319     #renameToCentimeters()
320     #adjustBytesName()
321     #adjustCentimetersName()
322     #adjustFrequencyName()
323     # VoltageDivider directory its easier to adjust manually

```

```
324     #generateJsons()
325     #compileJsons()
326     openJson()
327     #deleteJsons()
```

F.2 CÓDIGO DE EXEMPLO DE RECUPERAÇÃO DE DADOS SIMPLES

Código F.2 – Exemplo de recuperação de dados

```
1# Open root Json for analysis
2def openJson():
3    with open(f'{rootDirectory}/result.json') as resultJson:
4        #data = json.loads(line) for line in resultJson
5        dict = json.loads(resultJson.read())
6        print(len(dict))
7        for data in dict:
8            if(data['voltageDivider'] == '01k01k'
9                and data['bytes'] == '0100B'
10               and data['distance'] == '00.0cm'):
11                outputResultSearch = ''
12                outputResultSearch += f'{data["frequency"]}'
13                outputResultSearch += f' {data["overallTransmissions"
14                  "]}"'
15                outputResultSearch += f' {data["resultTransmission"]}'
16                print(outputResultSearch)
17            resultJson.close()
```

F.3 CÓDIGO DE EXEMPLO DE RECUPERAÇÃO DE DADOS DETALHADO

Código F.3 – Exemplo de recuperação de dados

```
1# Open root Json for analysis
2def openJson():
3    with open(f'{rootDirectory}/result.json') as resultJson:
4        #data = json.loads(line) for line in resultJson
5        dict = json.loads(resultJson.read())
6        print(len(dict))
7        voltageDivider = '01k01k'
8        bytes = '0100B'
9        distance = '04.0cm'
10       print(f'{voltageDivider} > {bytes} > {distance}')
11       for data in dict:
12           if(data["voltageDivider"] == voltageDivider
13              and data["bytes"] == bytes
14              and data["distance"] == distance):
15               outputResultSearch = ''
16               outputResultSearch += f'{data["frequency"]}'
17               if (len(data["cleanTransmissions"])) == 1:
18                   outputResultSearch += f' 0{data[""
19                     "cleanTransmissions"]}'
20               else:
21                   outputResultSearch += f' {data["cleanTransmissions
22                     "]}'
```

```
22             outputResultSearch += f' {data["'
23                 dirtyTransmissions"]}''
24         else:
25             outputResultSearch += f' {data["dirtyTransmissions"
26                 "]}"'
27         if (len(data["failedTransmissions"])) == 1):
28             outputResultSearch += f' {data[""
29                 failedTransmissions"]}''
30         outputResultSearch += f' {data["overallTransmissions"
31                 "]}"'
32         outputResultSearch += f' {data["resultTransmission"]}'
33         print(outputResultSearch)
34     resultJson.close()
```