



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Renan Kendy Fukuda Tomisaki

**Desenvolvimento de Interface Gráfica em Python para Controle de Potência
Real**

Florianópolis
2022

Renan Kendy Fukuda Tomisaki

**Desenvolvimento de Interface Gráfica em Python para Controle de Potência
Real**

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Engenharia Elétrica.

Orientador: Prof. André Luís Kirsten, Dr.

Florianópolis
2022

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

Tomisaki, Renan Kendy Fukuda
Desenvolvimento de interface gráfica em Python para
controle de potência real Florianópolis / Renan Kendy
Fukuda Tomisaki ; orientador, André Luís Kirsten, 2022.
111 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Engenharia Elétrica, Florianópolis, 2022.

Inclui referências.

1. Engenharia Elétrica. 2. Raspberry Pi. 3. Protocolos
de comunicação. 4. Python. 5. I2C. I. Kirsten, André Luís.
II. Universidade Federal de Santa Catarina. Graduação em
Engenharia Elétrica. III. Título.

Renan Kendy Fukuda Tomisaki

Desenvolvimento de Interface Gráfica em Python para Controle de Potência Real

O presente trabalho em nível de Bacharelado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. André Luís Kirsten, Dr.
Universidade Federal de Santa Catarina

Prof. Fábio Antônio Xavier, Dr.
Universidade Federal de Santa Catarina

Prof. Samir Ahmad Mussa, Dr.
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Bacharel em Engenharia Elétrica.

Prof. Miguel Moreto, Dr.
Coordenador do Programa

Prof. André Luís Kirsten, Dr.
Orientador

Florianópolis, 12 de julho de 2022.

Dedico este trabalho de conclusão de curso aos meus pais, que mesmo com a distância física sempre estiveram presentes na minha jornada acadêmica.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais que além de fornecerem mais do que o necessário para que eu pudesse aproveitar as oportunidades, sempre participaram ativamente em minha formação pessoal, apoiando nos momentos de dificuldade e celebrando ao meu lado nos momentos de felicidade.

Agradeço também aos meus colegas de curso e amigos que aconselharam nos momentos de dificuldade, incentivaram nas decisões tomadas e foram essenciais para a conclusão da graduação. Em especial, agradeço a Marina, por sua companhia e incentivo que vieram nos momentos em que mais precisava.

Aos professores do curso de engenharia elétrica, que se tornaram uma referência pessoal dentro e fora das salas de aula. Ao professor André Kirsten tenho uma gratidão especial por auxiliar na execução deste trabalho, mas também por apresentar o mundo da programação de uma perspectiva objetiva e intrigante.

“A persistência é o caminho do êxito.”
(Charles Chaplin)

RESUMO

O objetivo central deste trabalho é realizar o controle de potência fornecida a um equipamento ao controlar a corrente elétrica e realizar as medidas de corrente e tensão. Para realizar este controle, foi utilizado um dos menores computadores do mundo, o Raspberry Pi 4, para mostrar uma interface gráfica programada em Python que gera níveis de corrente de acordo com parâmetros previamente definidos pelo usuário. As informações são transmitidas através do protocolo de comunicação serial I2C até um microcontrolador que executa um código programado em C, sendo que as leituras dos valores de corrente e tensão do sistema são transmitidas do microcontrolador para o Raspberry e mostrados em um gráfico em tempo real. Uma das funcionalidades presentes no Raspberry Pi 4 é a conexão VNC que permite o compartilhamento de tela com outros computadores, possibilitando o controle e monitoramento remoto. Este projeto foi elaborado com o intuito de controlar a potência fornecida para uma fonte de solda, mas outros aparelhos também podem ser controlados com este mesmo projeto.

Palavras-chave: Python, C, Raspberry Pi, IoT, I2C.

ABSTRACT

The main objective of this work is to control the power supplied to an equipment by controlling the electric current and performing current and voltage measurements. To achieve this objective, one of the smallest computers in the world, the Raspberry Pi 4, was used to display a graphic user interface programmed in Python that generates current levels according to parameters previously defined by the user. The information is transmitted through the I2C serial communication protocol to a microcontroller which executes a code programmed in C, the readings of the system's current and voltage values are transmitted from the microcontroller to the Raspberry and shown in a real-time graph. One of the features present in the Raspberry Pi 4 is the VNC connection that allows screen sharing with other computers, enabling remote control and monitoring. This project was made with the intention of controlling the power supplied to a welding machine, but other devices can also be controlled with this same project.

Keywords: Python. C. I2C. Raspberry Pi. IoT.

LISTA DE FIGURAS

Figura 1 – Gráfico da transmissão de corrente.	25
Figura 2 – Representação das ligações entre os equipamentos.	26
Figura 3 – Exemplo de comunicação serial e paralela.	29
Figura 4 – Exemplo de transmissão e recepção simultâneas do protocolo <i>Universal Asynchronous Receiver/Transmitter</i> (UART).	31
Figura 5 – Exemplo de pacote de dados utilizado no protocolo UART.	31
Figura 6 – Modos de funcionamento do protocolo <i>Serial Peripheral Interface</i> (SPI). .	33
Figura 7 – Modos de comunicação do protocolo SPI de acordo com o <i>Clock Polarity</i> (CPOL) e <i>Clock Phase</i> (CPHA).	35
Figura 8 – Esquema de ligação dos resistores de <i>pull-up</i> nas linhas SDA e SCL. .	37
Figura 9 – Janela de tempo para mudança do nível lógico da SDA em relação à SCL.	37
Figura 10 – Janelas de tempo para mudança do nível lógico do início e fim da mensagem.	38
Figura 11 – Exemplo do formato de mensagem I2C.	39
Figura 12 – Funcionamento da sincronização de clock no I2C.	40
Figura 13 – Funcionamento da sincronização de clock no I2C.	41
Figura 14 – Modelo de camadas estabelecidas pelo ISO 11898.	42
Figura 15 – Formato de mensagem CAN com identificador de 11 bits.	43
Figura 16 – Formato de mensagem CAN com identificador de 29 bits.	44
Figura 17 – Foto ilustrativa do Raspberry Pi 4 model B.	53
Figura 18 – Acesso remoto do Raspberry Pi 4 pelo VNC Viewer em computador na mesma rede.	55
Figura 19 – Foto ilustrativa do microcontrolador.	56
Figura 20 – Pinos do microcontrolador.	56
Figura 21 – Registradores do módulo I2C no microcontrolador.	57
Figura 22 – Diagrama de funcionamento da comunicação com o microcontrolador.	63
Figura 23 – Configuração inicial para criar um novo projeto.	64
Figura 24 – Janela de confirmação da conexão entre microcontrolador e computador.	65
Figura 25 – Alterações nos pinos de comunicação I2C do arquivo F2806x_I2C.c.	67
Figura 26 – Diagrama de fluxo da recepção de dados pelo microcontrolador. .	68
Figura 27 – Diagrama de fluxo da transmissão de dados pelo microcontrolador. .	69
Figura 28 – Diagrama de fluxo das funções do RPi.	70
Figura 29 – Janela inicial do programa com as caixas de valores em branco. .	71
Figura 30 – Diagrama de fluxo da alteração dos parâmetros pelo usuário. . . .	72
Figura 31 – Reta do ponto A até o ponto B.	73

Figura 32 – Gráfico da corrente transmitida pelo tempo.	74
Figura 33 – Diagrama de fluxo da corrente enviada para o alvo.	75
Figura 34 – Diagrama de fluxo da corrente recebida pelo o controlador.	76
Figura 35 – Exemplo de curva gaussiana.	77
Figura 36 – Exemplo de dez mil amostras geradas utilizando a distribuição normal.	78
Figura 37 – Diagrama de fluxo da tensão recebida pelo o controlador.	79
Figura 38 – Diagrama de fluxo de atualização do gráfico da comunicação.	80
Figura 39 – Detecção de endereço do alvo.	82
Figura 40 – Parâmetros para iniciar a transmissão.	83
Figura 41 – Gráfico da corrente transmitida, corrente recebida e tensão recebida pelo tempo.	84
Figura 42 – Gráfico das potências e diferença entre a potência medida e a transmitida.	85
Figura 43 – Forma de onda de dados da linha SDA.	85
Figura 44 – Início da comunicação na linha SDA.	86

LISTA DE TABELAS

Tabela 1 – Comparação entre comunicação paralela e serial	30
Tabela 2 – Exemplo de bit de paridade	32
Tabela 3 – Sinais da comunicação SPI	34
Tabela 4 – Modos de comunicação SPI	36
Tabela 5 – Comparação entre os protocolos de comunicação serial	46
Tabela 6 – Descrição dos bits do registrador de funcionamento I2CMDR	58
Tabela 7 – Descrição dos bits do registrador de status I2CSTR	60
Tabela 8 – Descrição dos bits do registrador de interrupções I2CIER	61
Tabela 9 – Arquivos necessários na importação do CCS	66

LISTA DE ABREVIATURAS E SIGLAS

ACK *Acknowledge*

Bd *Baud*

CAN *Controller Area Network*

CCS *Code Composer Studio*

CI *Circuito Integrado*

CPHA *Clock Phase*

CPOL *Clock Polarity*

CSMA/CD *Carrier Sense Multiple Access with Collision Detection*

CSV *Comma Separated Values*

GPIO *General Purpose Input/Output*

GUI *Graphical User Interface*

I2C *Inter-Integrated Circuit*

ISO *International Standards Organization*

LSB *Least Significant Byte*

MSB *Most Significant Byte*

NACK *Not Acknowledge*

OSI *Open Systems Interconnection*

RPi *Raspberry Pi*

RTR *Remote Transmit Request*

SCL *Serial Clock*

SDA *Serial Data*

SPI *Serial Peripheral Interface*

UART *Universal Asynchronous Receiver/Transmitter*

SUMÁRIO

1	INTRODUÇÃO	25
1.1	OBJETIVOS GERAIS	25
1.1.1	Objetivos Específicos	26
1.2	METODOLOGIA	27
1.3	ESTRUTURA DO TRABALHO	27
2	EMBASAMENTO TEÓRICO	29
2.1	PROTOCOLOS DE COMUNICAÇÃO	29
2.1.1	<i>Universal Asynchronous Receiver/Transmitter - UART</i>	30
2.1.1.1	Bit de início	31
2.1.1.2	Bits de dados	32
2.1.1.3	Bit de paridade	32
2.1.1.4	Bit de parada	32
2.1.2	<i>Serial Peripheral Interface - SPI</i>	33
2.1.2.1	Modo controlador	34
2.1.2.2	Modo alvo	34
2.1.2.3	Modos de transmissão	34
2.1.3	<i>Inter-Integrated Circuit - I2C</i>	36
2.1.3.1	Linhas de comunicação	36
2.1.3.2	Validação dos dados	37
2.1.3.3	Início e Pausa	37
2.1.3.4	Formato dos dados	38
2.1.3.5	ACK e NACK	38
2.1.3.6	Arbitragem e sincronização de clock	39
2.1.3.7	Endereço do alvo	41
2.1.3.7.1	<i>Bit de leitura ou escrita</i>	41
2.1.4	<i>Controller Area Network - CAN</i>	41
2.1.4.1	Tipos de mensagens	43
2.1.4.2	Mensagem de dados	44
2.1.4.3	Mensagem remota	45
2.1.4.4	Mensagem de erro	45
2.1.4.5	Mensagem de sobrecarga	45
2.1.5	Comparação entre os protocolos	45
2.2	LINGUAGENS DE PROGRAMAÇÃO	46
2.2.1	Programação no nível de máquina	46
2.2.2	Programação de baixo nível (<i>Assembly</i>)	47
2.2.3	Programação de alto nível	48
2.2.3.1	Linguagem de programação C	48

2.2.3.2	Python	49
2.2.3.2.1	<i>NumPy</i>	49
2.2.3.2.2	<i>pandas</i>	50
2.2.3.2.3	<i>matplotlib</i>	50
2.2.3.2.4	<i>tkinter</i>	51
3	EQUIPAMENTOS DO PROJETO	53
3.1	RASPBERRY PI	53
3.1.1	Especificações do equipamento	54
3.1.2	Instalação do sistema operacional	54
3.1.3	Acesso remoto	54
3.2	PICCOLO CONTROLSTICK TMS320F28069	55
3.2.1	C2000Ware	57
3.2.2	Code Composer Studio	57
3.2.3	Protocolo I2C no microcontrolador	57
3.2.3.1	Registrador de recepção de dados I2CDRR	58
3.2.3.2	Registrador de transmissão de dados I2CDXR	58
3.2.3.3	Registrador de modo de funcionamento I2CMDR	58
3.2.3.4	Registrador de endereço no modo alvo I2COAR	59
3.2.3.5	Registrador de contagem de dados I2CCNT	59
3.2.3.6	Registrador de estado I2CSTR	60
3.2.3.7	Registrador de habilitação das interrupções I2CIER	61
4	DESENVOLVIMENTO	63
4.1	FUNCIONAMENTO DO MICROCONTROLADOR COMO O ALVO DO PROJETO	63
4.1.1	Configurações do microcontrolador	64
4.1.2	Arquivos do projeto no microcontrolador	66
4.1.3	Microcontrolador no modo alvo receptor	67
4.1.4	Microcontrolador no modo alvo transmissor	68
4.2	FUNCIONAMENTO DO RASPBERRY PI COMO CONTROLADOR DO PROJETO	70
4.2.1	Configuração dos parâmetros	70
4.2.1.1	Lista de tempo	72
4.2.1.2	Lista dos valores de corrente	73
4.2.1.3	Gráfico da corrente transmitida	74
4.2.2	Reset dos parâmetros	74
4.2.3	Teste de conexão com o alvo	75
4.2.4	Transmissão de corrente	75
4.2.5	Recepção de corrente	76
4.2.5.1	Distribuição normal	77

4.2.6	Recepção de tensão	78
4.2.7	Gráfico em tempo real da comunicação	79
4.2.8	Análise de arquivos	80
4.3	CONEXÃO ENTRE O RASPBERRY PI E O MICROCONTROLADOR	81
5	SIMULAÇÕES	83
6	CONCLUSÃO	87
	REFERÊNCIAS	89
	APÊNDICE A – ARQUIVO EXAMPLE_2806XI2C_EEPROM.C DO C2000WARE COM AS ALTERAÇÕES	91
	APÊNDICE B – ARQUIVO EM PYTHON DO RASPBERRY PI	99

1 INTRODUÇÃO

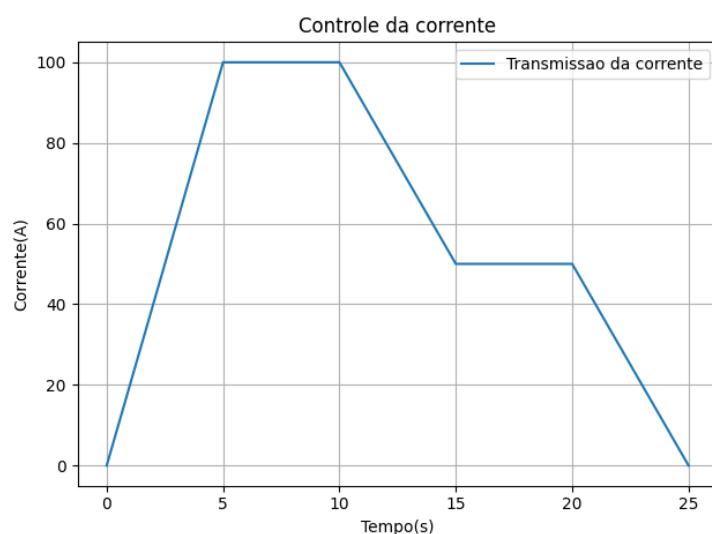
Existem muitas maneiras de controlar a potência real fornecida à uma carga, desde soluções analógicas até o controle de maneira digital. Esses métodos de controle da potência na carga cumprem suas funções ao regular a corrente elétrica, alterar a tensão fornecida ou mudando o valor de carga no circuito.

Além do controle da potência, é interessante comparar os valores de referência enviados com os valores reais do sistema para assegurar de que os resultados obtidos sejam correspondentes aos parâmetros definidos previamente. Outro ponto importante é armazenar os dados da potência desejada e da potência medida no sistema, para que ao comparar os parâmetros e resultados obtidos a configuração otimizada seja encontrada, e também para encontrar os parâmetros das configurações que não entregam o resultado desejado.

1.1 OBJETIVOS GERAIS

Neste projeto deseja-se controlar a potência entregue a uma máquina de solda ao utilizar níveis de referência de corrente gerados de acordo com as especificações do usuário por meio de uma interface gráfica de software programado em Python como mostra a figura 1. O software mostra os níveis de referência que estão sendo enviados, realiza a leitura dos valores de corrente e tensão no sistema e mostra-os em um gráfico em tempo real.

Figura 1 – Gráfico da transmissão de corrente.



Fonte: Autoria própria

Outra função do programa é armazenar os parâmetros e resultados em arquivos

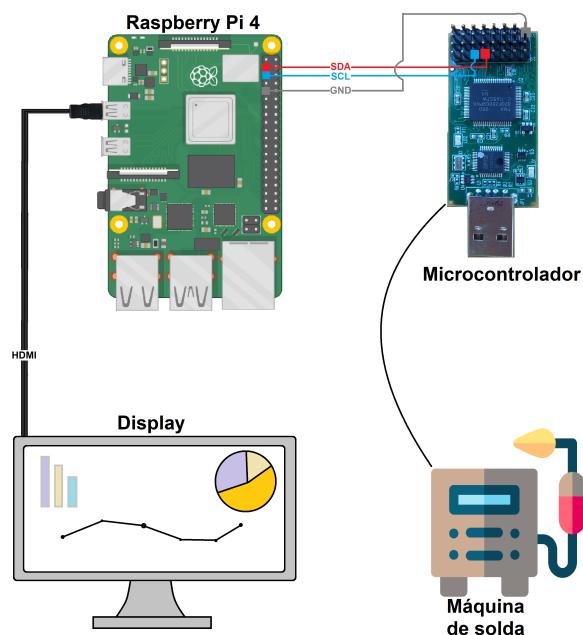
que possam ser compartilhados e analisados em outras plataformas. O formato utilizado para a gravação é o CSV, que separa as categorias de informação por vírgulas.

1.1.1 Objetivos Específicos

- Revisar os protocolos de comunicação mais relevantes;
- Apresentar o computador Raspberry Pi 4 model B e o microcontrolador Piccolo controlSTICK;
- Apresentar as linguagens de programação utilizadas;
- Implementar os softwares e a comunicação serial nos equipamentos;
- Simular o controle da potência de um equipamento utilizando este projeto;

A Figura 2 mostra os componentes utilizados e suas ligações, onde o Raspberry Pi é conectado com a tela por um cabo HDMI, e com o microcontrolador por meio de três cabos jumpers e a máquina de solda deve ser ligado ao microcontrolador, mas neste trabalho os dados que seriam recebidos da leitura da máquina de solda serão simulados pelo microcontrolador e processados no Raspberry Pi.

Figura 2 – Representação das ligações entre os equipamentos.



Fonte: Compilação de imagens dos sites Raspberry Pi Foundation, CNX Software e Flaticon.

1.2 METODOLOGIA

Este trabalho possui caráter de pesquisa exploratória, onde a pesquisa bibliográfica mostra os protocolos de comunicação mais utilizados, funcionamento dos equipamentos e linguagens de programação utilizadas. A abordagem possui característica quantitativa na utilização dos dados contidos no embasamento teórico com o intuito de desenvolver uma solução para o controle da potência elétrica.

O planejamento de estudo, análise de caso e execução do trabalho consistem nas etapas apresentadas a seguir:

- Protocolos de comunicação;
- Linguagens de programação;
- Aparelhos eletrônicos utilizados;
- Implementação dos códigos nos equipamentos;
- Simulação de comunicação entre os equipamentos;

1.3 ESTRUTURA DO TRABALHO

O segundo capítulo deste trabalho apresenta o embasamento teórico utilizado para desenvolver este projeto, apresentando alguns dos métodos para comunicação entre aparelhos e apresenta as características principais das linguagens de programação utilizadas.

O terceiro capítulo aborda as características mais importantes dos componentes eletrônicos utilizados, explica também algumas configurações necessárias para a utilização dos mesmos.

No quarto capítulo é apresentado como as informações do segundo e terceiro capítulos se aplicam para cada etapa do projeto, quais linguagens de programação são utilizadas em quais equipamentos e o funcionamento destes softwares para que possam comunicar entre si através de um barramento de comunicação serial.

O funcionamento do ponto de vista de um usuário do projeto é mostrado no quinto capítulo, configurando as informações necessárias no software e como transmiti-las para realizar o controle.

O último capítulo faz um resumo dos resultados obtidos na simulação, conclusão e sugestões de possíveis alterações.

2 EMBASAMENTO TEÓRICO

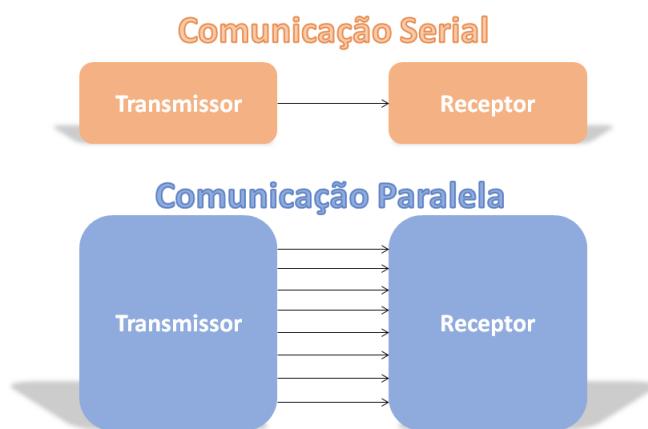
Neste capítulo serão apresentados os fundamentos teóricos utilizados na elaboração deste projeto, as linguagens de programação e também serão apresentados os equipamentos com algumas de suas funções relevantes.

2.1 PROTOCOLOS DE COMUNICAÇÃO

Muitos dos produtos eletrônicos são compostos de diversos circuitos integrados que podem ser conectados na montagem final, este método é essencial para a produção em massa, pois permite que os diferentes módulos sejam fabricados simultaneamente. Para que o produto final funcione corretamente é necessário que os circuitos integrados comuniquem-se entre si, para este propósito existem dois principais protocolos de comunicação, são eles: a comunicação serial e a comunicação paralela (RF WIRELESS WORLD, 2012).

Na comunicação paralela o pacote de bits é enviado em bloco, ou seja, todos os bits são enviados ao mesmo tempo de um Circuito Integrado (CI) para outro em um ciclo de *clock*, como representado na parte inferior da Figura 3. O tamanho do bloco de dados a ser enviado depende do número de linhas de transmissão, para transmitir um bloco de 8 bits são necessários no mínimo 8 linhas (RF WIRELESS WORLD, 2012).

Figura 3 – Exemplo de comunicação serial e paralela.



Fonte: Adaptado de (ETIQ TECHNOLOGIES, 2016)

A comunicação serial consiste na transferência de dados serialmente um após o outro, sendo que o processamento dos dados é realizado de forma independente da transferência dos mesmos, possibilitando a comunicação entre processadores sem a existência de uma memória compartilhada (MANKAR *et al.*, 2014).

Pelo fato de os pacotes de dados serem enviados bit a bit, a velocidade de transmissão é um fator importantíssimo e pode ser calculada pela Equação 1 onde T_{bps} é a velocidade de transmissão em bit por segundo e t_b é o tempo necessário para a transmissão de 1 bit (RF WIRELESS WORLD, 2012).

$$T_{bps} = \frac{1}{t_b} \quad (1)$$

Existem vantagens e desvantagens para a utilização destes métodos de comunicação. Nos casos em que existe a necessidade de alta velocidade de transmissão, a comunicação paralela é a melhor opção, mas para os casos em que a velocidade de transmissão não é o fator decisivo e sim o custo, então a comunicação serial é a melhor opção. A Tabela 1 compara as vantagens e desvantagens das duas formas de comunicação.

Tabela 1 – Comparação entre comunicação paralela e serial

Especificações	Paralela	Serial
Número de bits que podem ser transmitidos em um ciclo de clock	n bits	1 bit
Número de linhas necessárias para transmitir n bits	n linhas	1 linha
Velocidade de transferência	Rápida	Lenta
Custo de transmissão	Alto	Baixo
Aplicação	Necessidade de alta velocidade de transmissão em que o custo não é a prioridade	Prioridade em redução de custo e economia de espaço físico

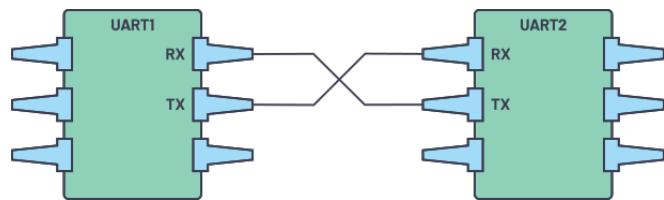
Fonte: Adaptado de RF Wireless World (2012)

2.1.1 Universal Asynchronous Receiver/Transmitter - UART

A comunicação UART por definição é assíncrona, ou seja, a transmissão e recepção dos dados podem ser realizadas independente uma da outra e não requerem um sinal de *clock*, caracterizando uma comunicação *full duplex* onde a transmissão e recepção de dados pode ser feita ao mesmo tempo. Vale notar que é possível trabalhar somente com a recepção ou transmissão. Somente um fio é necessário para a transmissão e outro para a recepção como mostra a Figura 4, por conta da falta do sinal de *clock* na comunicação, é obrigatório que a velocidade em que o UART transmissor e o UART receptor trabalham sejam iguais, ambos os aparelhos precisam alterar seus

ciclos de *clock* internos para uma mesma velocidade, esta configuração é definida pelo *baud rate*. O *baud rate* é a quantidade máxima de símbolos por segundo que podem ser enviados na linha de comunicação e é medido em Baud (Bd). A diferença de *baud rate* do transmissor para o receptor não pode ultrapassar dez por cento para manter a comunicação funcionando corretamente (PEÑA; LEGASPI, 2020).

Figura 4 – Exemplo de transmissão e recepção simultâneas do protocolo UART.



Fonte: Peña e Legaspi (2020)

A transferência de dados é realizada por meio de pacotes que podem variar de sete a treze bits no total como mostra a figura 5. Sendo eles divididos em um bit para a sinalização do início de transmissão, cinco a nove bits contendo os dados de informação, de um a dois bits para sinalizar o término da transmissão e opcionalmente um bit de paridade (PEÑA; LEGASPI, 2020).

Figura 5 – Exemplo de pacote de dados utilizado no protocolo UART.

Bit de inicio 1 bit	Bits de dados 5 a 9 bits	Bit de paridade 0 ou 1 bit	Bit de parada 1 a 2 bits
------------------------	-----------------------------	-------------------------------	-----------------------------

Fonte: Adaptado de Peña e Legaspi (2020)

2.1.1.1 Bit de início

Quando o UART transmissor está em modo de espera ele mantém o barramento em nível lógico alto, na maioria dos casos, e sinaliza o início da transmissão mantendo o nível lógico baixo durante um ciclo de *clock*. Assim que o UART receptor percebe a alteração de tensão no barramento ele começa a fazer a leitura na velocidade do *Baud rate* escolhido. Caso o modo de espera do UART transmissor seja no nível lógico baixo, o início da transmissão será feito quando o barramento for para o nível lógico alto durante um ciclo de *clock*.

O bit de início da Figura 4 representa o bit sinalizador de início da transmissão onde somente um bit é utilizado nesta função.

2.1.1.2 Bits de dados

Os bits de dados são mostrados na Figura 5 como bits de dados, referindo-se aos dados de informação de fato que podem conter de 5 até 8 bits se for utilizado o bit de paridade tem no máximo 9 bits de informação caso o bit de paridade não exista.

Na maioria dos casos os bits de dados começam a transmissão pelo bit menos significativo.

2.1.1.3 Bit de paridade

Os bits de paridade da Figura 4 representa o bit de paridade, que tem como função principal assegurar de que não houve erro durante a transmissão dos bits de dados, os erros podem ocorrer por radiação eletromagnética, velocidades de transmissão diferentes ou por conta de longas distâncias de transferência de dados.

Depois que o UART faz a leitura dos bits de dados, ele conta quantos números um (nível lógico alto) estão presentes no pacote de dados, quando essa soma é igual a um número ímpar o bit de paridade é igual a um (nível lógico alto) e é igual a zero (nível lógico baixo) quando a soma é um número par.

Na Tabela 2 é exemplificada a diferença de bit de paridade par e ímpar, o número 7 na base binária possui três níveis lógicos altos na base binária, o bit de paridade neste caso é igual a zero. O número 3 na base decimal possui dois níveis lógicos altos na base binária, portanto o bit de paridade é igual a um.

Tabela 2 – Exemplo de bit de paridade

Número em base decimal	Número em base binária	Quantidade de dígitos um	Bit de paridade
7	0000 0111	3	0
3	0000 0011	2	1

Fonte: Autoria própria

Quando a leitura do bit de paridade difere do número de bits com nível lógico alto, sabe-se que ocorreu um erro na comunicação e aumenta a confiabilidade quando o bit de paridade está em concordância com a soma níveis lógicos altos.

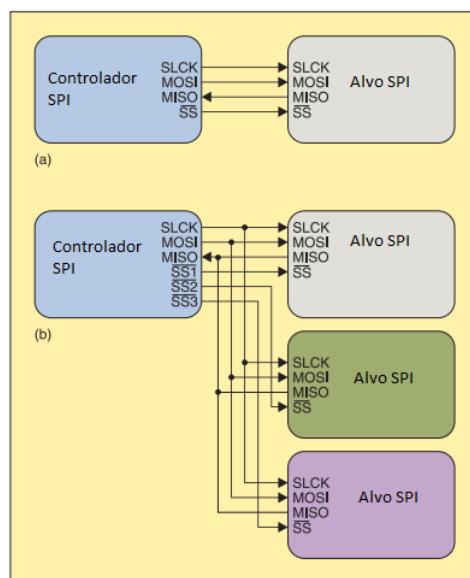
2.1.1.4 Bit de parada

O bit de parada é representado pelo bit de parada na Figura 4, este bit sinaliza o fim da transmissão utilizando de um a dois ciclos de clock no nível lógico alto.

2.1.2 Serial Peripheral Interface - SPI

O SPI é um protocolo de comunicação que permite a comunicação entre um SPI controlador e um alvo, assim como entre um controlador e múltiplos alvos como mostrado, em (a) e (b) respectivamente, na Figura 6 (LEENS, 2009).

Figura 6 – Modos de funcionamento do protocolo SPI.



Fonte: Adaptado de Leens (2009)

O protocolo de comunicação SPI foi introduzido pela Motorola com o lançamento de microcontrolador com arquitetura derivada do Motorola 6800 lançado em 1979 e utilizado nos computadores Macintosh da Apple em 1984 (LEENS, 2009). O módulo SPI permite uma comunicação serial, síncrona e duplex, ou seja, diferente do UART ele necessita de um sinal de *clock* para a sincronização e o controlador pode receber e transmitir informação simultaneamente (MOTOROLA, 2004).

Os quatro sinais necessários na comunicação SPI são descritos, com suas respectivas funções, na Tabela 3

Tabela 3 – Sinais da comunicação SPI

Sinal	Descrição
SCLK	<i>Serial clock</i> é o sinal enviado do controlador para que todos os alvos comuniquem de maneira sincronizada.
SSn	<i>Slave select signal</i> é o sinal para selecionar o alvo com o qual o controlador quer estabelecer a comunicação.
MOSI	<i>Master Out-Slave In</i> é a linha de transmissão de dados do controlador para o alvo.
MISO	<i>Master In-Slave Out</i> é a linha de recepção do controlador, onde o alvo consegue enviar dados para o controlador.

Fonte: Adaptado de (LEENS, 2009)

2.1.2.1 Modo controlador

No modo controlador, somente o SPI controlador pode iniciar a transmissão. O início da transmissão ocorre quando a linha SSn é alterada do nível lógico alto para o baixo, o *clock* na linha SCLK vai para uma frequência em que o controlador e o alvo selecionado possam trabalhar, somente então os dados são transmitidos na linha MOSI sobre o controle do SCLK (LEENS, 2009).

2.1.2.2 Modo alvo

No modo alvo o módulo SPI tem o seu pino SCLK funcionando como uma entrada que utilizará o *baud rate* gerado pelo controlador. Antes que a transmissão seja iniciada o sinal no pino SSn tem que estar em nível lógico baixo e manter-se assim até o fim da transmissão, caso o SSn vá para o nível lógico alto durante a transmissão o SPI é forçado a ir para o modo de espera (MOTOROLA, 2004).

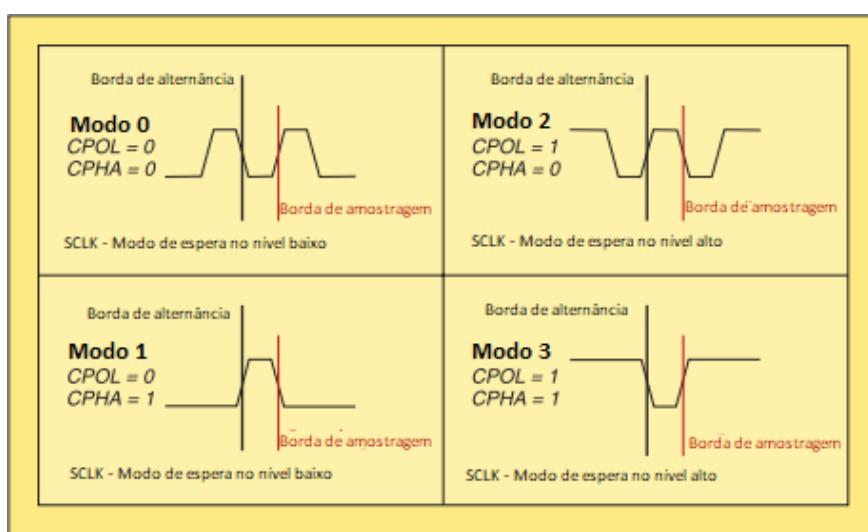
2.1.2.3 Modos de transmissão

Existem quatro formas de transmissão que podem ser selecionadas de acordo com os bits referentes ao CPOL e CPHA. O CPOL refere-se à polaridade do sinal de *clock* durante o estado de espera e não possui um efeito significativo no formato da transmissão (MOTOROLA, 2004). O estado de espera é definido como o período entre a transição de nível lógico alto para o baixo do SSn no início da transmissão e durante a transição do nível lógico baixo para o alto do SSn no final da transmissão (DHAKER, 2018).

O CPHA seleciona a fase do *clock*, dependendo do valor do seu bit pode ser utilizada a borda de subida ou descida do *clock* para amostrar ou para mover os bits de dados para a linha. Os modos de comunicação são mostrados com suas respectivas configurações de CPOL e CPHA na Tabela 4 e na Figura 7 também é possível visualizar a diferença na transmissão de dados de acordo com os modos de comunicação (DHAKER, 2018).

Para o funcionamento correto do módulo SPI é necessário que o controlador e alvo possuam as mesmas configurações de CPOL, CPHA e *clock*, no caso de um sistema com alvos com configurações diferentes, o controlador precisa se reconfigurar de modo que possa ser reconhecido pelo alvo desejado (LEENS, 2009).

Figura 7 – Modos de comunicação do protocolo SPI de acordo com o CPOL e CPHA.



Fonte: Fonte: Adaptado de Leens (2009)

Tabela 4 – Modos de comunicação SPI

Modo	CPOL	CPHA	Polaridade do <i>clock</i>	Fase de <i>clock</i> utilizada para amostragem e/ou mover dados
0	0	0	Nível lógico baixo	Dados amostrados na borda de subida e movidos na borda de descida
1	0	1	Nível lógico baixo	Dados amostrados na borda de descida e movidos na borda de subida
2	1	1	Nível lógico alto	Dados amostrados na borda de descida e movidos na borda de subida
3	1	0	Nível lógico alto	Dados amostrados na borda de subida e movidos na borda de descida

Fonte: Adaptado de (DHAKER, 2018)

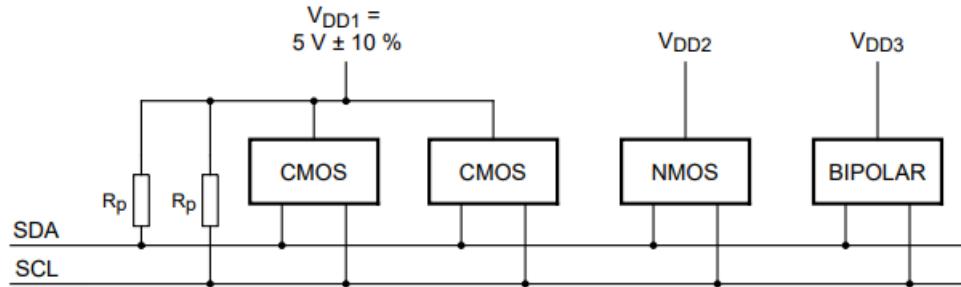
2.1.3 *Inter-Integrated Circuit* - I2C

O protocolo de comunicação *Inter-Integrated Circuit* (I2C) é um protocolo utilizado mundialmente, mais de cinquenta empresas e mais de mil fabricantes de circuitos integrados utilizam esta tecnologia em seus produtos (NXP SEMICONDUCTORS, 2021). Nos laboratórios da Philips em Eindhoven, nos países baixos, foi criado o I2C com o objetivo de facilitar a comunicação entre os periféricos das televisões nos anos 80. A velocidade máxima da época era de cem quilo bits por segundo, conhecido como a velocidade padrão, em 1995 a velocidade rápida de quatrocentos quilo bits foi estabelecida e desde 1998 a velocidade alta de transmissão supera três mega bits por segundo. Atualmente a empresa NXP Semiconductors, antiga Philips Semiconductors, é a responsável pela regularização das especificações do protocolo (LEENS, 2009).

2.1.3.1 Linhas de comunicação

A informação é transmitida entre os aparelhos pelo fio *Serial Data* (SDA). Já o clock é definido pelo fio *Serial Clock* (SCL), esses são os dois únicos fios necessários para esta comunicação *half-duplex*, que é uma comunicação onde o alvo e controlador podem enviar dados um para o outro, porém não o fazem simultaneamente. As linhas de SDA e SCL são linhas bidirecionais que devem estar conectadas em uma fonte de tensão positiva ou com resistores de *pull-up* como mostra a Figura 8. Os valores de referência para os níveis lógicos alto e baixo são, respectivamente, setenta por cento do valor do V_{DD} e trinta por cento do V_{DD} (NXP SEMICONDUCTORS, 2021).

Figura 8 – Esquema de ligação dos resistores de *pull-up* nas linhas SDA e SCL.

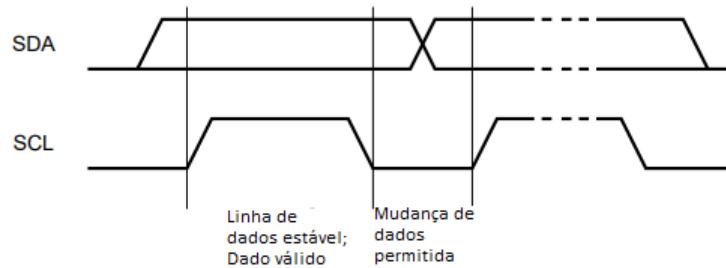


Fonte: NXP Semiconductors (2021)

2.1.3.2 Validação dos dados

Para que os dados da linha SDA sejam validados, a mudança entre níveis lógicos deve ocorrer enquanto a linha SCL está no nível lógico baixo e a SDA deve terminar a alteração de nível lógico antes que a SCL inicie a subida de nível. A Figura 9 ilustra os momentos em que os níveis devem ser alterados para que os dados sejam computados corretamente.

Figura 9 – Janela de tempo para mudança do nível lógico da SDA em relação à SCL.



Fonte: Adaptado de NXP Semiconductors (2021)

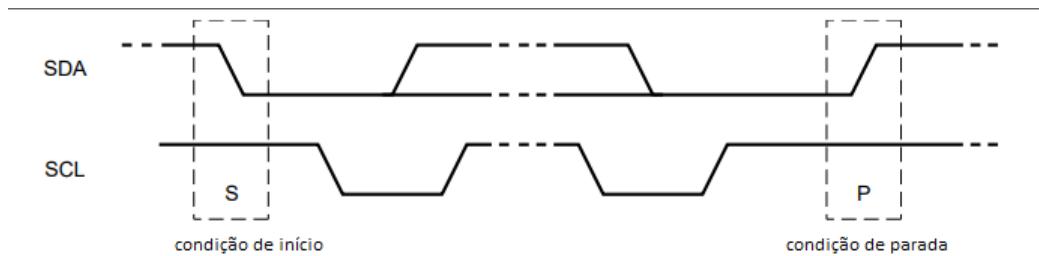
2.1.3.3 Início e Pausa

O controlador é quem determina quando mensagens podem ser enviadas na linha de SDA, quando nenhum dado está sendo transmitido a linha SDA se mantém no nível lógico alto. Quando o controlador vai iniciar uma mensagem, ele espera o momento em que a linha SCL está no nível lógico alto, muda o nível lógico da linha SDA do alto para o baixo e termina esta transição antes da linha SCL ir para o nível lógico baixo.

O sinal de parada do controlador para o alvo é similar ao início da mensagem, com a diferença de que a linha SDA vai passar do nível lógico baixo para o alto enquanto a SCL está no nível lógico alto. A Figura 10 mostra o momento em que a

mensagem inicia, representado pela letra S, e momento em que se encerra, representado pela letra P.

Figura 10 – Janelas de tempo para mudança do nível lógico do início e fim da mensagem.



Fonte: Adaptado de NXP Semiconductors (2021)

Quando o controlador quer enviar outra mensagem, ao invés de enviar um sinal de parada, ele envia um sinal de início repetido, que como o nome sugere é um sinal onde a linha SDA vai passar do nível lógico alto para o baixo durante o ciclo de nível alto da SCL (NXP SEMICONDUCTORS, 2021).

2.1.3.4 Formato dos dados

Após o envio do sinal de início da mensagem, o mestre envia os dados da mensagem que possuem oito bits. O número de bytes que podem ser enviados em uma transferência é ilimitado, mas entre os bytes o alvo deve enviar um sinal de *Acknowledge* (ACK) para confirmar o recebimento do byte. Enquanto o receptor não terminou de processar os dados, ele mantém o nível lógico baixo do clock para forçar o controlador a esperar que ele esteja pronto para receber ou enviar outro byte. A transferência dos bytes é iniciada pelo bit mais significativo, ou seja, da esquerda para a direita do número (NXP SEMICONDUCTORS, 2021).

2.1.3.5 ACK e NACK

O ACK é o bit de confirmação do alvo para o transmissor, confirmando se o receptor conseguiu receber o byte de dados. Para que o receptor confirme que o byte foi recebido completamente, o transmissor cede o controle da linha SDA no nível lógico alto e o receptor altera o nível para baixo (NXP SEMICONDUCTORS, 2021).

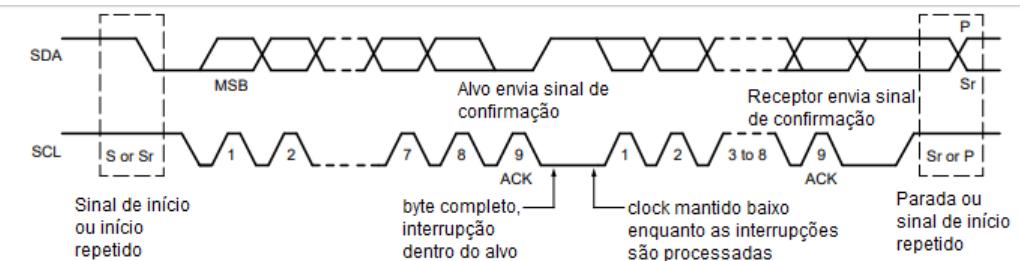
O *Not Acknowledge* (NACK) é chamado assim pois representa o não aparecimento do ACK, ou seja, quando a linha SDA não é puxada para o nível baixo no nono

ciclo de clock após o início da mensagem. Existem cinco motivos que podem causar este problema, são eles:

- O controlador não conseguiu acertar o endereço do alvo para a comunicação;
- O alvo não estava pronto para receber a mensagem do controlador;
- Durante a transferência o alvo recebeu informações das quais não comprehende;
- O alvo recebeu mais bytes do que suporta;
- Quando controlador recebeu dados e quer sinalizar o fim da transmissão, ele envia um NACK para o alvo transmissor.

A Figura 11 representa as etapas da transmissão de mensagens do protocolo I2C.

Figura 11 – Exemplo do formato de mensagem I2C.



Fonte: Adaptado de NXP Semiconductors (2021)

2.1.3.6 Arbitragem e sincronização de clock

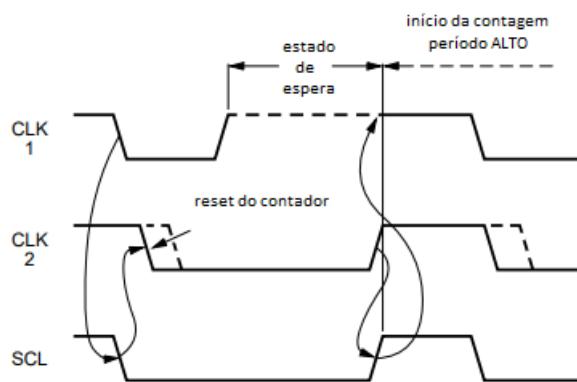
A rede de comunicação no protocolo I2C pode possuir mais de um controlador, mas para que a comunicação ocorra corretamente é necessário realizar a sincronização de clock e levar em consideração a arbitragem entre os controladores. Tanto a sincronização de clock e arbitragem não são aplicáveis para redes com um único controlador.

Os ciclos de clock na linha SCL são definidos pela sincronização de clock dos controladores, onde o controlador com o maior tempo no nível lógico baixo determina o tempo em que a SCL fica no nível baixo e o controlador com o tempo de nível lógico alto mais curto irá determinar a duração do nível alto na linha SCL. Isso acontece pela operação booleana *AND* que ocorre entre a ligação dos controladores com a SCL, onde o primeiro controlador que colocar o seu clock no nível baixo irá alterar a SCL para o nível baixo e assim será mantido até que ambos os controladores passem o nível lógico de seus próprios clocks para o nível alto. Quando o controlador que possui

a menor duração de nível lógico baixo de clock está esperando o outro controlador subir o nível lógico do clock, o primeiro controlador mantém o nível lógico alto até que o outro controlador suba o clock, fazendo com que a duração de um ciclo completo de clock, alto e baixo, dure o mesmo tempo para os controladores.

Na Figura 12 é possível notar que durante o tempo de espera do CLK1, linha tracejada do *wait state*, ele mantém o nível alto e só inicia a duração do nível lógico alto de clock quando o CLK2 também sobe o próprio nível (NXP SEMICONDUCTORS, 2021).

Figura 12 – Funcionamento da sincronização de clock no I2C.

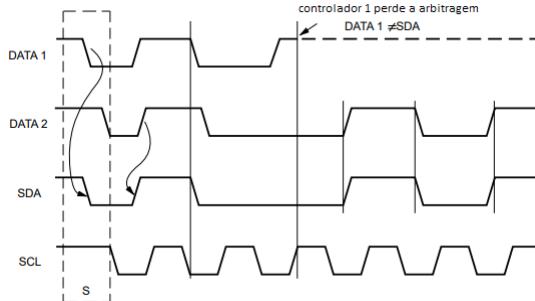


Fonte: Adaptado de NXP Semiconductors (2021)

Para que o controlador em uma rede com mais de um controlador envie uma mensagem, ele verifica por um curto espaço de tempo se a rede está livre, então envia um sinal de início de mensagem na linha SDA, caso mais de um controlador inicie a transmissão ao mesmo tempo é possível que ambos enviem a mesma mensagem, portanto não haverá problema. O controlador ao enviar um bit, verifica qual o valor lido na linha SDA, caso os valores sejam os mesmos ele irá enviar o próximo bit, isso é extremamente útil na arbitragem.

A arbitragem é feita de acordo com o nível lógico do bit que está sendo enviado pelos controladores, caso ambos os controladores tenham iniciado a transmissão ao mesmo tempo, irão enviar o bit e realizar a leitura da SDA, quando um dos controladores percebe que a leitura está diferente do envio, caso que ocorre somente quando um dos controladores envia o nível lógico baixo e outro o nível alto. O controlador que enviou o bit alto e realizou a leitura da SDA no nível baixo, identifica que perdeu a arbitragem e cede o controle da SDA, o controlador espera o fim da mensagem para que possa reiniciar a transmissão (NXP SEMICONDUCTORS, 2021). A Figura 13 ilustra como funciona a arbitragem da linha SDA com dois controladores.

Figura 13 – Funcionamento da sincronização de clock no I2C.



Fonte: NXP Semiconductors (2021)

2.1.3.7 Endereço do alvo

Assim como pode existir mais de um controlador na rede, pode haver mais de um alvo e para que o controlador consiga se comunicar especificamente com um alvo, o alvo precisa de uma identificação, conhecida como o seu endereço. Dentro dos endereços existem dois formatos:

- Endereço de sete bits, permitindo até cento e vinte sete aparelhos na rede.
- Endereço de dez bits, permitindo até mil e vinte três aparelhos na rede.

2.1.3.7.1 Bit de leitura ou escrita

Logo após o bit de início da mensagem, o controlador envia o endereço de qual alvo com quem deseja se comunicar. Como a linha SDA é uma linha bidirecional, o controlador precisa especificar se deseja que o alvo receba uma mensagem ou que transmita para o controlador, para isso é enviado o bit de escrita logo após o endereço do alvo. Quando o bit de escrita é igual a um, significa que o controlador deseja enviar uma mensagem para o alvo, quando é igual a zero, o controlador deseja que o alvo envie uma mensagem para o controlador (NXP SEMICONDUCTORS, 2021).

2.1.4 Controller Area Network - CAN

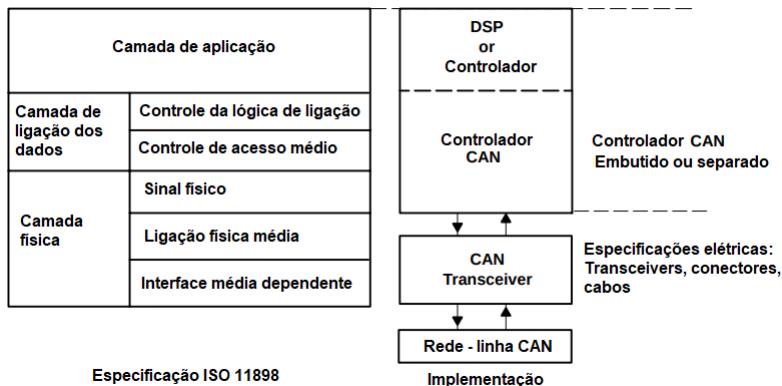
O protocolo de comunicação *Controller Area Network* (CAN) foi criado no meio dos anos 80 pela empresa alemã Robert Bosch para aplicações em automóveis, visando uma comunicação serial robusta para oferecer automóveis mais seguros e eficientes na utilização de combustível, e ao mesmo tempo diminuir o peso e a complexidade dos cabos de comunicação (PAZUL, 2002). Com o CAN, a complexidade de cabos nos automóveis foi reduzida a uma rede de comunicação com dois fios,

velocidades máximas de um milhão de bits por segundo, alta imunidade contra interferência elétrica e habilidade de autodiagnosticar e corrigir erros nos dados (CORRIGAN, 2002).

Diferente do USB, a rede CAN não envia um pacote grande de informação do ponto A até o ponto B sobre a supervisão de um controlador, ela envia várias mensagens curtas para toda a rede, o que permite uma maior consistência na comunicação (CORRIGAN, 2002). Cabe a cada nó do sistema decidir se a informação recebida deve ser guardada ou apagada (PAZUL, 2002).

Para permitir a interoperabilidade entre produtos de companhias diferentes, a organização não governamental *International Standards Organization* (ISO) criou o manual de referência *Open Systems Interconnection* (OSI) que define o modelo em termos de camadas (PAZUL, 2002). O protocolo de comunicação CAN, ISO 11898, descreve como a informação é passada entre os aparelhos conectados na rede, padronizando a conexão entre a camada física (*Physical Layer*) e a camada de dados (*Data-Link Layer*) do modelo OSI/ISO mostrados na Figura 14.

Figura 14 – Modelo de camadas estabelecidas pelo ISO 11898.



Fonte: Corrigan (2002)

O protocolo CAN é um *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD). CSMA significa que antes de enviar uma mensagem, todos os nós no sistema precisam monitorar a rede por um período de tempo e verificar a ausência de mensagens sendo transmitidas. Ao final deste período todos os nós possuem a mesma oportunidade de enviar uma mensagem, por isso o nome de múltiplos acessos (PAZUL, 2002). O protocolo CAN possui também uma detecção de colisão que no caso de dois nós enviarem uma mensagem ao mesmo tempo, o nó com a maior prioridade ganha o controle da rede, sendo que essa prioridade é definida durante o projeto do sistema (CORRIGAN, 2002). Outra funcionalidade do CAN é o *Remote Transmit Request* (RTR), requisição de transmissão remota, que ao invés de esperar para enviar

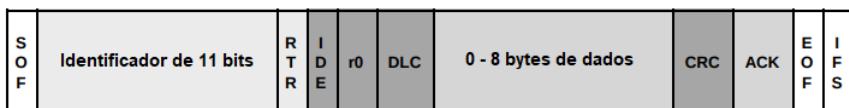
uma mensagem, um nó pode requisitar a transmissão de dados de outro nó (PAZUL, 2002).

O protocolo CAN é uma versão de baixa velocidade normatizado pela ISO 11519 com transmissão de até 125 kbps. O CAN 2.0A da norma ISO 11898:1993 com velocidade de 1 Mbps e o CAN 2.0B da norma ISO 11898:1995 com 1 Mbps. O CAN de baixa velocidade e o CAN 2.0A possuem identificadores de 11 bits, o CAN 2.0B possui identificador de 29 bits (CORRIGAN, 2002).

2.1.4.1 Tipos de mensagens

A mensagem do CAN padrão (CAN de baixa velocidade e CAN 2.0A) é separada por em onze campos diferentes que são mostrados na Figura 15.

Figura 15 – Formato de mensagem CAN com identificador de 11 bits.



Fonte: Corrigan (2002)

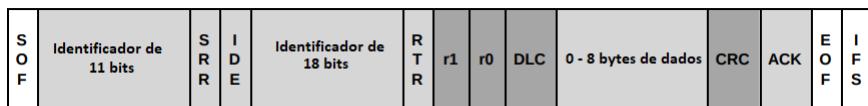
Os significados dos campos são:

- SOF (*Single dominant start Of Frame*) é o bit que sinaliza o início da mensagem e é utilizado para sincronizar os nós da rede.
- Identificador de 11 bits, possibilitando a identificação de 2048 mensagens diferentes.
- RTR (*Remote Transmission Request*) este bit é responsável por mostrar se o nó está requisitando.
- IDE (*Identifier extension*) este bit serve para sinalizar se o identificador possui uma extensão.
- r0 (*Reserved bit*) este bit está disponível para uma possível alteração na norma.
- DLC (*Data Length Code*) possui 4 bits representando a quantidade de bytes de dados sendo transmitidos.
- Dados podendo conter até 8 bytes (64 bits) de informação.
- CRC (*Cyclic Redundancy Check*) possui 16 bits para checar se ocorreu algum erro na transmissão, contendo o número de bits já enviados.

- ACK (*Acknowledgement*) serve para o nó transmissor receber um sinal dos nós receptores caso a transmissão esteja acontecendo corretamente. O bit é enviado recessivo, o nó receptor que recebeu a mensagem corretamente altera este bit para um bit dominante, comunicando para o nó transmissor que a mensagem foi recebida, mas quando o bit permanece recessivo significa que nenhum dos nós receptores recebeu a mensagem corretamente, neste caso um sinal de erro é gerado e o nó espera uma nova arbitragem para enviar a mensagem novamente.
- EOF (*End Of Frame*) campo com 7 bits que marca o final da mensagem, nestes sete bits o nó transmissor indica se a transmissão foi bem sucedida ou não.
- IFS (*Inter-Frame Space*) são 7 bits para dar tempo ao controlador mover a mensagem recebida para a posição correta.

O CAN 2.0B possui um formato ligeiramente diferente, além dos 18 bits identificadores adicionais existem outros bits mostrados na Figura 16 e também o bit do IDE é recessiva para sinalizar que existe uma extensão na mensagem (CORRIGAN, 2002).

Figura 16 – Formato de mensagem CAN com identificador de 29 bits.



Fonte: Corrigan (2002)

Os bits adicionais são:

- SRR *Substitute Remote Request* que substitui o RTR da mensagem do CAN padrão;
- r1 é outro bit reserva.

Uma mensagem é considerada livre de erros caso os bits do EOF sejam totalmente recessivos, um bit dominante significa que houve um erro e a transmissão é reiniciada (CORRIGAN, 2002).

2.1.4.2 Mensagem de dados

A mensagem de dados é o tipo de mensagem mais comum, é constituída pelos campo de arbitragem, de dados e de reconhecimento. O campo de arbitragem determina a prioridade da mensagem. No CAN 2.0A ele possui 11 bits de identificação e o bit RTR é dominante para mostrar que os dados serão enviados pelo nó. No CAN

2.0B possui 29 bits de identificação e o RTR. Depois vem o campo de dados que pode variar de 0 a 8 bytes de informação, seguido pelo CRC de 16 bits usados para informar o número de bits já transmitidos. Por último vem o bit de reconhecimento, caso os bits recebidos coincidam com o CRC, então o bit de reconhecimento passa de recessivo para dominante (CORRIGAN, 2002).

2.1.4.3 Mensagem remota

A mensagem remota é muito similar à mensagem de dados, mas com a diferença do bit RTR que é recessivo, exatamente para sinalizar que o nó está requisitando informação de outro. A outra diferença é a de que não são enviados dados (CORRIGAN, 2002).

2.1.4.4 Mensagem de erro

A mensagem de erro é especial pois viola o formato padrão das mensagens do CAN. Ela é transmitida por um nó quando encontra um erro na mensagem e faz com que os outros nós da rede também enviem um sinal de erro, fazendo com que o nó transmissor reinicie o envio da mensagem (CORRIGAN, 2002).

2.1.4.5 Mensagem de sobrecarga

A mensagem de sobrecarga tem características similares da mensagem de erro com relação ao formato, e é transmitida por um nó quando ele está sobrecarregado. Este tipo de mensagem é usado principalmente para dar um tempo de espera entre as mensagens (CORRIGAN, 2002).

2.1.5 Comparação entre os protocolos

A Tabela 5 compara as vantagens dos protocolos, vale a pena notar que dependendo da aplicação certos pontos considerados como vantagem podem ser considerados como uma desvantagem.

Tabela 5 – Comparaçāo entre os protocolos de comunicação serial

Protocolo	Vantagens	Desvantagens
UART	Simples implementação, popular, não precisa de sinal de clock.	Funcionalidade limitada, somente dois aparelhos se comunicam, baixa velocidade.
SPI	Rápido, implementação de baixo custo.	Não possui um padrão fixo, utiliza muitos pinos de saída, somente um controlador.
I2C	Simples, bom custo benefício, múltiplos controladores e alvos.	Número de componentes é limitado por conta da capacitância, pode se tornar complexo com muitos alvos.
CAN	Seguro, rápido.	Complexo, orientado a automóveis.

Fonte: Adaptado de Mankar *et al.* (2014)

Todos os protocolos atendem aos requisitos mínimos do projeto, porém os protocolos UART e SPI possuem um número limitado de alvos, tornando a expansão do projeto mais limitada. Apesar de o protocolo CAN possuir um número grande de componentes na rede, ele é o mais complexo na execução e as suas velocidades de comunicação são muito superiores às velocidades necessárias. O protocolo I2C possui o melhor custo benefício, oferecendo um grande número de alvos para serem conectados na rede, boas velocidades de transmissão e possuir uma implementação mais simples.

2.2 LINGUAGENS DE PROGRAMAÇÃO

Os computadores operam de acordo com um conjunto de regras e instruções para que executem uma tarefa. Para controlar o sistema do computador é necessária uma sequência de conjuntos gramaticais, também conhecidas como linguagens de programação. Dentre as linguagens de programação é possível dividi-las em três grupos de acordo com a sua utilização (YUVAYANA, 2015).

2.2.1 Programação no nível de máquina

Assim como o nome sugere, a programação no nível de máquina é a linguagem do computador, esta é a forma de programação mais baixo nível. Consiste em de somente duas condições, verdadeiro (1 ou nível lógico alto) ou falso (0 ou nível lógico baixo). A programação no nível de máquina foi a primeira geração das linguagens de programação, onde todas as instruções dadas ao computador têm que estar em números binários, ou seja, zero ou um (YUVAYANA, 2015).

Algumas das vantagens de utilizar a programação no nível de máquina são:

- A linguagem vai interagir diretamente com o computador.
- Não existe a necessidade de programas como compiladores ou interpretadores.
- Leva pouco tempo para executar os programas, pois não exige conversão de linguagens.

Algumas das desvantagens de utilizar a programação no nível de máquina são:

- Cada máquina requer instruções diferentes.
- Alta dificuldade para compreensão e construção de programas.
- Dificuldade em encontrar erros no código.

2.2.2 Programação de baixo nível (*Assembly*)

Os primeiros computadores custavam milhares de dólares, enquanto um bom programador custava cerca de quinze mil dólares por ano. Devido a essa disparidade nos custos, do ponto de vista econômico, era mais vantajoso que um programador fizesse a programação em nível de máquina, mas com o passar do tempo o preço dos computadores foi caindo e o dos programadores aumentando, fazendo com que uma programação mais rápida se tornasse vantajosa. Foi neste período que surgiu a segunda geração das linguagens de programação, onde o próprio computador traduzia para a linguagem de máquina (OUALLINE, 1997).

A linguagem *Assembly* é a segunda geração das linguagens de programação, ela contém as mesmas instruções da programação no nível de máquina, porém as instruções e variáveis possuem nomes próprios ao invés de utilizar somente os números binários. Alguns símbolos também são usados nessas linguagens, principalmente para descrever campos de instruções como os ponteiros. A linguagem *Assembly* utiliza uma linha por comando de instrução da máquina, com o auxílio de um *assembler* esses comandos são traduzidos para números binários, na linguagem de máquina, e assim executados em ordem (YUVAYANA, 2015).

Algumas das vantagens de utilizar *Assembly* para a programação são:

- Facilidade para a compreensão por humanos devido a conversão para números binários ser realizado por compilador.
- A programação leva menos tempo devido a facilidade na expressão de ideias.
- Erros são mais fáceis de serem encontrados e corrigidos.

Algumas das desvantagens de utilizar *Assembly* para a programação são:

- É uma linguagem dependente de cada máquina na qual será programada, pois cada máquina utiliza um conjunto de comandos diferentes.
- Alguns comandos podem ser de difícil compreensão.

Por ser uma das primeiras formas de programação, o Assembly assemelha-se muito com a linguagem de máquinas, com a criação de outras linguagens de programação, atualmente o Assembly pode ser considerado como uma linguagem no nível de máquina, pois não faz o uso de um compilador para executar o programa.

2.2.3 Programação de alto nível

A terceira geração das linguagens de programação é chamada de alto nível, ela é assim chamada pois as linguagens nesta categoria se assemelham com a linguagem humana, portanto é mais facilmente interpretada pelos usuários. Para que o computador possa executar os comandos dados pelo usuário, um compilador ou interpretador se faz necessário. Existem várias linguagens de alto nível, como o FORTRAN, C, C++, C#, Python, entre outros (YUVAYANA, 2015).

Algumas das vantagens de utilizar a programação de alto nível são:

- Os comandos e instruções possuem uma maior facilidade de compreensão pelo programador.
- A lógica e estrutura do código são mais fáceis de compreender.
- Menor tempo de programação.
- Programas podem ser utilizados em máquinas diferentes sem necessidade de alteração no código.

Algumas das desvantagens de utilizar a programação de alto nível são:

- A execução dos programas é mais lenta.
- Programação de alto nível ocupa mais espaço quando comparada com as gerações anteriores.

2.2.3.1 Linguagem de programação C

A linguagem de programação C foi desenvolvida durante os anos de 1969-1973, mas foi padronizada pelo comitê ANSI X3J11 somente nos anos 80. Ela ganhou este nome pois foi derivada da linguagem de programação B (RITCHIE, 1993). O C foi desenvolvido com o intuito de ser a ponte entre o programador e a máquina. Os programas de computador podem ser divididos entre dados e instruções, existe uma diferença clara para o programador, mas não para o computador (OUALLINE, 1997).

Com a evolução da memória de armazenamento e processamento das tecnologias, as linguagens de programação foram evoluindo junto, porém a linguagem C não evolui tanto quanto outras que foram surgindo, sendo considerada por muitos como uma linguagens de baixo nível ou a linguagem de nível mais baixo dentre as linguagem de alto nível.

Ao utilizar C é possível definir os dados em números inteiros, fracionários, positivos, negativos, etc. O próprio compilador se encarregará de converter para o nível de máquina, mas é necessário que ele saiba como aquela instrução em inglês se traduz. Para que a tradução ocorra de acordo com o esperado, basta que no início do código a biblioteca correta seja utilizada (OUALLINE, 1997).

Em C é possível também fazer o uso de arquivos diferentes em conjunto, caso queira executar uma instrução em um novo arquivo basta utilizar o arquivo antigo com a instrução desejada, evitando a necessidade de o programador escrever o mesmo código em arquivos diferentes (OUALLINE, 1997).

2.2.3.2 Python

Desde o seu surgimento em 1991, o Python se tornou uma das linguagens de programação mais populares. Sua política de código aberto possibilitou a participação da comunidade da computação científica em seu desenvolvimento contínuo, causando um aumento no uso desta linguagem em aplicações industriais e acadêmicas (MCKINNEY, 2017).

Existem várias bibliotecas disponíveis para o Python, algumas das principais para a análise de dados são o NumPy, pandas e matplotlib (MCKINNEY, 2017). A biblioteca do tkinter é extremamente útil para desenvolver uma *Graphical User Interface* (GUI) (interface gráfica do usuário).

Os sistemas operacionais geralmente não vêm o Python instalado de fábrica. A instalação do Python é simples, realizada através do terminal de comando ao inserir `sudo apt install python3 idle3`. O Python possui várias versões, porém não existe compatibilidade entre a versão dois e a três, neste projeto foi utilizado a versão 3.7.3 do Python.

2.2.3.2.1 NumPy

NumPy ou Numerical Python é o pacote de base para a computação científica (MCKINNEY, 2017), algumas das funções que ele provém são:

- Uma matriz multidimensional rápida e eficiente.
- Funções para trabalhar com elementos específicos da matriz e também operações matemáticas entre matrizes.

- Operações de álgebra, transformadas de Fourier e geração de números aleatórios.
- Ferramentas para integrar códigos em C, C++ e Fortran com o Python.

O NumPy não vem instalado com o Python, para instalar é necessário utilizar o comando `pip install numpy` no terminal.

2.2.3.2.2 *pandas*

A biblioteca do pandas facilita muito quando o objetivo é analisar e manipular dados, pois integra o processamento de alta performance do NumPy com a flexibilidade de manipulação das tabelas (MCKINNEY, 2017).

Outra funcionalidade do pandas é a importação de arquivos em *Comma Separated Values* (CSV) (valores separados por vírgula), tipo de arquivo onde os dados são separadas vírgulas, e são convertidas em *DataFrames*. *DataFrame* é uma estrutura de dados tabular, bidimensional e orientada pelas colunas com rótulos para colunas e linhas (MCKINNEY, 2017).

O pandas não vem instalado com o Python, para instalar é necessário utilizar o comando `pip install pandas` no terminal.

2.2.3.2.3 *matplotlib*

A biblioteca do Python mais utilizada para gerar gráficos é o matplotlib, onde é possível gerar diferentes tipos de gráficos e entre eles estão:

- Gráfico de pizza.
- Gráfico de linhas.
- Gráfico de barras.
- Gráfico de dispersão.

É possível também fazer a junção destes tipos gráficos em uma mesma imagem, além de ser possível ampliar a imagem em regiões de interesse e é possível também mover a imagem para outras áreas mantendo a proporção de ampliação (MCKINNEY, 2017).

O matplotlib não vem instalado com o Python, a instalação é feita utilizando o comando `pip install matplotlib` no terminal.

2.2.3.2.4 *tkinter*

O tkinter permite gerar uma interface gráfica para que o usuário possa interagir com o programa de forma simples oferecendo uma forma de adicionar menus, botões, caixa de diálogo, etc.

Caso o Python não venha com uma instalação do tkinter, o comando `pip install tk` no terminal realiza a instalação.

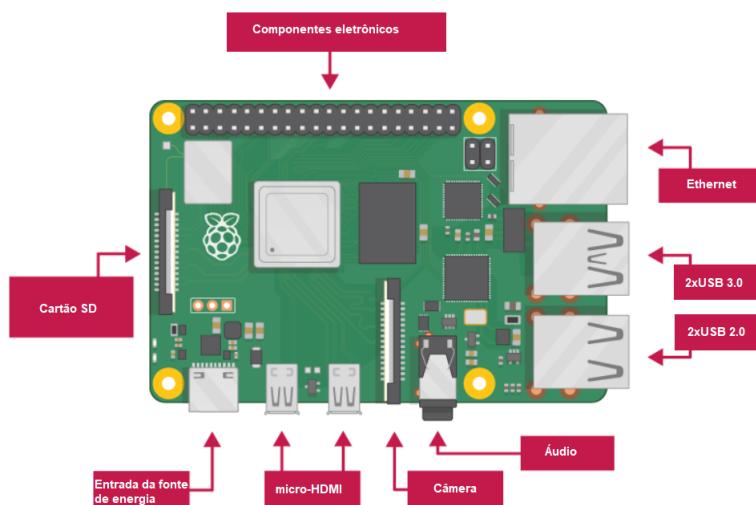
3 EQUIPAMENTOS DO PROJETO

Nesta seção serão apresentadas algumas das características mais importantes dos equipamentos utilizados neste projeto. Os equipamentos utilizados são altamente flexíveis e podem ser aplicadas em uma enorme gama de projetos, portanto somente as características relevantes ao projeto serão apresentadas.

3.1 RASPBERRY PI

O *Raspberry Pi* (RPi) foi desenvolvido com o intuito de ser um computador barato para que as novas gerações pudessem aprender o básico da programação. O primeiro RPi foi lançado em 2012 e até o ano de 2019 foram vendidas mais de dezenove milhões de unidades (GHAEL; SOLANKI; SAHU, 2020). A Figura 17 ilustra o RPi utilizado neste projeto.

Figura 17 – Foto ilustrativa do Raspberry Pi 4 model B.



Fonte: Adaptado de (RASPBERRY PI FOUNDATION, 2020)

Além de ser um computador barato e rápido, o RPi possui vinte e seis pinos de entrada e saída, *General Purpose Input/Output* (GPIO), aumentando a gama de aplicações onde o RPi pode ser aplicado, algumas delas são:

- Automação residencial.
- Servidor web.
- Câmera de segurança.
- Monitor de qualidade do ar.

- Repetidor de sinal de Wi-Fi.

Muitos dos projetos citados podem ser criados utilizando os GPIOs do RPi, mas que funcionem em conjunto com outros periféricos é necessária a utilização de um protocolo de comunicação.

3.1.1 Especificações do equipamento

O RPi 4 utilizado neste projeto vem com:

- Processador Broadcom BCM2711 de quatro núcleos, Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz.
- 8GB de memória RAM.
- Wi-Fi de 2.4 GHz e 5 GHz.
- 2 portas USB 2.0 e 2 portas de USB 3.0.
- 2 portas de micro-HDMI.
- Bluetooth 5.0.

Os pinos padrão de I2C no RPi 4 são os pinos 3 e 5, são respectivamente, o SDA e SCL. Outros pinos podem ser utilizados como pinos I2C, mas para usá-los o RPi precisa ser configurado fora do seu padrão.

Além do próprio RPi é necessário um mouse, teclado, monitor e um cartão microSD de no mínimo oito giga bytes para instalar o sistema operacional.

3.1.2 Instalação do sistema operacional

O RPi pode executar vários sistemas operacionais baseados em Linux, o mais popular é o Raspbian que pode ser encontrado no site oficial do Raspberry Pi. Ao realizar o download do Raspberry Pi Imager e instalar o programa em um computador, inserir o microSD e seguir os passos para a instalação do sistema operacional no cartão, o RPi está pronto para o uso.

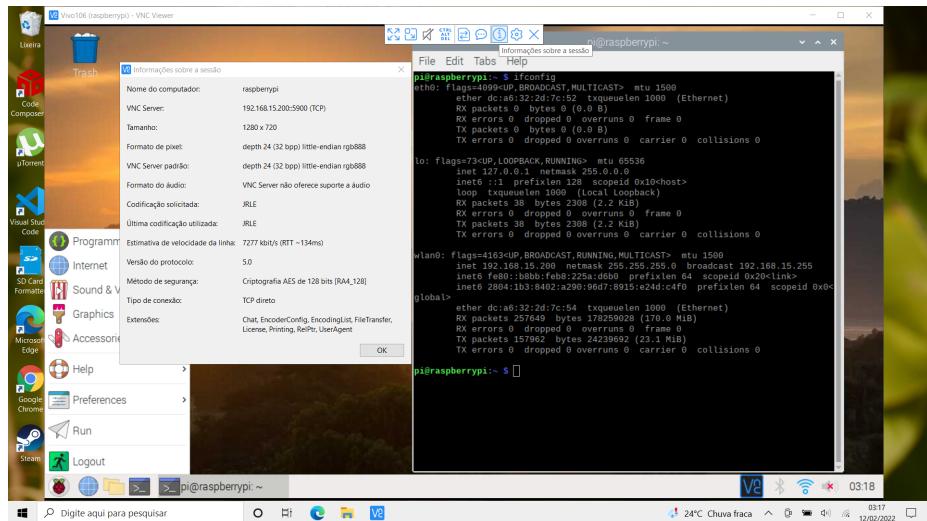
3.1.3 Acesso remoto

É possível utilizar o RPi por meio de outro computador conectado na mesma rede de internet. Com o sistema operacional instalado no RPi e conectado na internet, ao abrir a janela de comandos e digitar o comando `ifconfig` o endereço de IP será exibido. O próximo passo é habilitar a conexão VNC que pode ser feita seguindo os passos:

- Digitar na janela de comando `sudo raspi-config`
- Abrir a janela Interfacing Options
- Selecionar Yes na linha VNC e salvar as alterações.

O último passo é instalar o VNC Viewer no computador com o qual desejar controlar o RPi remotamente, iniciar uma nova conexão com o endereço de IP do RPi. Na primeira conexão o usuário é `pi` e a senha é `raspberry`. A Figura 18 mostra a tela do RPi sendo mostrada no programa VNC Viewer em um computador windows.

Figura 18 – Acesso remoto do Raspberry Pi 4 pelo VNC Viewer em computador na mesma rede.

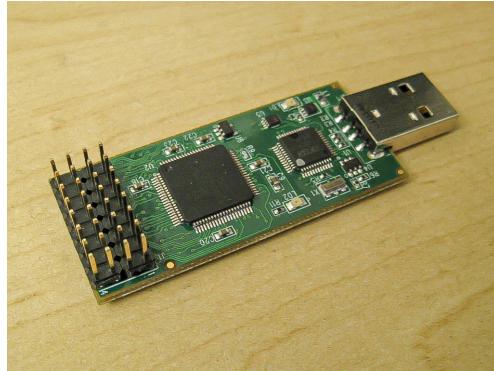


Fonte: Autoria própria

3.2 PICCOLO CONTROLSTICK TMS320F28069

O microcontrolador Piccolo controlSTICK TMS320F28069, mostrado na Figura 19 faz parte da família C2000, são microcontroladores produzidos pela Texas Instruments para realizar o controle em tempo real com baixa latência para aplicações industriais e automotivas (TEXAS INSTRUMENTS, 2017).

Figura 19 – Foto ilustrativa do microcontrolador.



Fonte: Adaptado de (COLTON, 2012)

Algumas de suas características são:

- CPU eficiente de 32 bits.
- Memória flash de 256 kBytes.
- Memória RAM de 100 kBytes.
- Portas seriais para diferentes protocolos de comunicação.
- 32 pinos de saída

A Figura 20 mostra quais os pinos de saída devem ser utilizados de acordo com o objetivo do projeto. Para este projeto os pinos 32 e 33, 28 e 29, são de maior interesse, pois são os pinos de SDA e SCL.

Figura 20 – Pinos do microcontrolador.

1 ADC-A6 COMP3(+VE)	2 ADC-A2 COMP1(+VE)	3 ADC-A0	4 3V3
5 ADC-A4 COMP2(+VE)	6 ADC-B1	7 EPWM-4B GPIO-07	8 TZ1 GPIO-12
9 SCLA GPIO-33	10 ADC-B6 COMP3(-VE)	11 EPWM-4A GPIO-06	12 ADC-A1
13 SDAA GPIO-32	14 ADC-B0	15 EPWM-3B GPIO-05	16 5V0 (Disabled by Default)
17 EPWM-1A GPIO-00	18 ADC-B4 COMP2(-VE)	19 EPWM-3A GPIO-04	20 SPISOMIA GPIO-17
21 EPWM-1B GPIO-01	22 ADC-A5	23 EPWM-2B GPIO-03	24 SPISIMOA GPIO-16
25 SPISTEA GPIO-19	26 ADC-B2 COMP1(-VE)	27 EPWM-2A GPIO-02	28 GND
29 SPICLKA GPIO-18	30 GPIO-34 (LED)	31 PWM1A-DAC (Filtered)	32 GND

Fonte: Adaptado de (TEXAS INSTRUMENTS, 2018)

3.2.1 C2000Ware

O microcontrolador F28069 é extremamente versátil e por conta disso muitos módulos necessitam de configuração. Para mitigar a dificuldade de sua aplicação para certos projetos, a Texas Instruments oferece o programa C2000Ware. O programa disponibiliza vários exemplos de aplicação, bibliotecas e ferramentas que facilitam a utilização dos módulos de hardware para os microcontroladores da série C2000. Os arquivos fornecidos pelo C2000Ware serviram de base para a programação do alvo. O instalador pode ser encontrado no site oficial da Texas Instruments.

3.2.2 Code Composer Studio

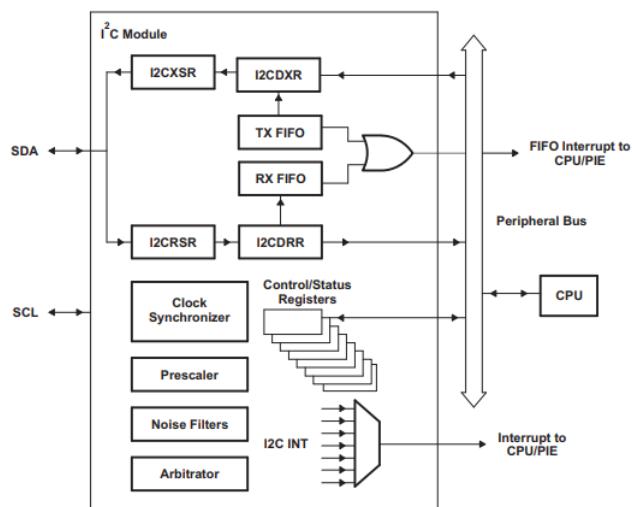
O *Code Composer Studio* (CCS) é o programa para desenvolvimento dos produtos da Texas Instruments, possui ferramentas úteis para desenvolver, compilar e depurar códigos em C e C++. Este programa pode ser adquirido de forma gratuita no site oficial da Texas Instruments. Neste projeto foi utilizada a versão 11.1.0.00011 do CCS.

3.2.3 Protocolo I2C no microcontrolador

O controlSTICK pode funcionar tanto como um controlador, quanto um alvo e pode transmitir ou receber em ambos os modos. Possui duas velocidades de transmissão, a padrão e a rápida. Possui compatibilidade com o smbus e endereçamento de sete e dez bits.

A Figura 21 mostra quais registradores estão entre o CPU e as linhas de SDA e SCL.

Figura 21 – Registradores do módulo I2C no microcontrolador.



Fonte: (TEXAS INSTRUMENTS, 2018)

3.2.3.1 Registrador de recepção de dados I2CDRR

O registrador de recepção de dados que pode ser acessado pela CPU é o I2CDRR. É um registrador de dezesseis bits, mas comporta até oito bits de dados e os outros oito bits restantes são reservados, não haverá alteração no funcionamento do registrador caso algum dado seja escrito nos oito bits reservados, somente os oito primeiros bits serão contabilizados.

Na Figura 21 o registrador I2CRSR entre o I2CDRR e a linha SDA, é o registrador de deslocamento da recepção e não possui ligação direta à CPU. Quando o registrador I2CRSR recebeu todos os bits ele move esses dados para o I2CDRR, onde a CPU terá acesso aos dados para que possa realizar as operações necessárias (TEXAS INSTRUMENTS, 2017).

3.2.3.2 Registrador de transmissão de dados I2CDXR

O registrador I2CDXR possui dezesseis bits no total, mas somente oito bits são utilizados para o envio de dados. Os bits que devem ser transmitidos são movidos da CPU para o I2CDXR e quando os dados foram completamente movidos o I2CXSR recebe os dados para transmitir na linha SDA (TEXAS INSTRUMENTS, 2017).

3.2.3.3 Registrador de modo de funcionamento I2CMDR

O registrador responsável por quais os modos de funcionamento do módulo I2C do microcontrolador serão utilizados é I2CMDR. A Tabela 6 descreve quais as funções dos dezesseis bits deste registrador.

Tabela 6 – Descrição dos bits do registrador de funcionamento I2CMDR

Bit	Nome	Descrição
15	NACKMOD	<p>Este bit só possui aplicações práticas quando o módulo I2C é o receptor.</p> <p>Quando este bit é igual a zero, o microcontrolador trabalhando como o alvo nunca irá enviar um bit de NACK e no modo controlador irá enviar o NACK quando o contador interno for igual a zero, mas não irá enviar em outras situações.</p> <p>Quando este bit é igual a um, em ambos os modos será enviado um bit de NACK na linha SDA quando estiver no ciclo de <i>acknowledgment</i>.</p>
14	FREE	<p>Este bit serve para que o microcontrolador continue com o prosseguimento do código quando a depuração do código estiver sendo executada.</p>
13	STT	<p>Este bit só tem importância quando o microcontrolador está no modo controlador. Quando este bit é igual a um, o microcontrolador gera um sinal de início de mensagem na linha SDA. Após o sinal de início gerado, este bit volta para o valor zero.</p>

Continua...

Bit	Nome	Descrição
12	Reservado	Bit reservado, qualquer alteração neste bit não surtirá efeito no funcionamento do módulo I2C.
11	STP	Este bit só tem importância quando o microcontrolador está no modo controlador. Quando este bit é igual a um, o microcontrolador gera um sinal de parada da mensagem na linha SDA. Depois de gerar o sinal de parada, este bit volta para o valor zero.
10	MST	O bit MST indica em qual modo o módulo I2C está trabalhando, quando este bit é igual a um o módulo I2C está no modo controlador e quando é igual a zero está no modo alvo.
9	TRX	Bit do modo transmissor, quando este bit é igual a um, o microcontrolador funciona como o transmissor na rede e quando este bit é igual a zero ele funciona como o receptor.
8	XA	<i>Expanded Adress</i> bit é igual a um quando a rede está trabalhando com endereçamento de dez bits e igual a zero quando o endereçamento é de sete bits.
7	RM	<i>Repeat Mode</i> bit é aplicável somente quando o módulo I2C é o controlador no modo transmissor. Quando igual a um, o microcontrolador envia as mensagens assim que o registrador I2CDXR recebe novos dados.
6	DLB	<i>Digital Loopback</i> bit é responsável por habilitar o modo de <i>loopback</i> do microcontrolador no modo controlador. Quando este bit é igual a um o registrador I2CDRR recebe o valor que o I2CDXR enviou depois de passarem alguns ciclos de clock interno.
5	IRS	<i>I2C Reset</i> bit serve para limpar os estados do registrador I2CSTR, voltando para os seus valores padrão. O reset só é habilitado quando o bit IRS do I2CMDR é igual a um, mas só terá efeito quando for igual a zero.
4	STB	O <i>Start Byte</i> bit só é aplicável quando o módulo I2C está no modo controlador. Quando este bit é igual a um o módulo I2C envia mais mensagens e espera o alvo enviar um sinal ACK para enviar a mensagem desejada.
3	FDF	O <i>Free Data Format</i> é habilitado quando é igual um, ele funciona sem utilizar endereços para a comunicação na rede.
0-2	BC	<i>Bit Count</i> são os bits que determinam quantos bits serão transmitidos ou recebidos na mensagem.

Adaptado de Texas Instruments (2017)

3.2.3.4 Registrador de endereço no modo alvo I2COAR

O registrador I2COAR possui dezesseis bits, mas somente dez bits são utilizados. Este registrador tem a função de guardar o próprio endereço de alvo, dependendo do modo de endereçamento selecionado no I2CMDR podem ser utilizados somente sete bits ou todos os dez (TEXAS INSTRUMENTS, 2017).

3.2.3.5 Registrador de contagem de dados I2CCNT

O registrador I2CCNT é um registrador de dezesseis bits que registra quantas mensagens devem ser transmitidas ou recebidas (TEXAS INSTRUMENTS, 2017).

3.2.3.6 Registrador de estado I2CSTR

O registrador de estado I2CSTR sinaliza qual o tipo de interrupção ocorreu, a Tabela 7 descreve quais as funções dos dezesseis bits deste registrador.

Tabela 7 – Descrição dos bits do registrador de status I2CSTR

Bit	Nome	Descrição
15	Reservado	É sempre igual a zero e alterações neste bit são ignoradas.
14	SDIR	<i>Slave Direction</i> bit é o bit que indica em qual sentido a linha SDA está. Quando este bit é igual a um significa que o controlador deseja que o microcontrolador transmita uma mensagem e quando é zero o microcontrolador não é um alvo receptor.
13	NACKSNT	<i>NACK Sent</i> bit só exerce uma função quando está operando no modo receptor. Quando este bit é igual a um significa que o microcontrolador enviou um bit de NACK na SDA
12	BB	<i>Bus Busy</i> bit indica que a SDA está ocupada, ou seja, outro componente está enviando uma mensagem na linha.
11	RSFULL	<i>Receive Shift register Full</i> é o bit representativo do estado do registrador I2CRSR. Quando este bit é igual significa que o I2CDRR ainda não conseguiu processar os dados e o I2CRSR recebeu um novo byte, impossibilitando a recepção de novos dados da linha SDA.
10	XSMT	<i>Transmit Shift register empty</i> é o bit que quando é igual a zero, indica que o registrador I2CXSR está vazio. Quando o I2CDXR recebe novos dados o bit XSMT recebe o valor um.
9	AAS	<i>Addressed-As-Slave</i> bit indica quando o microcontrolador recebeu uma requisição do controlador. Este bit já é igual a um quando ocorreu uma chamada geral ou reconheceu o seu próprio endereço de alvo.
8	AD0	<i>Address 0 bits</i> é igual a um quando uma chamada geral é enviada na rede.
7-6	Reservado	Bits reservados que não possuem efeito na comunicação.
5	SCD	<i>Stop Condition Detected</i> é o bit que sinaliza um bit de parada na rede. Quando este bit é igual a um, significa que o controlador requisitou o encerramento da mensagem na rede.

Continua...

Bit	Nome	Descrição
4	XRDY	O sinalizador <i>Transmit-Data-Ready</i> serve para indicar quando o microcontrolador está pronto para transmitir mensagens para o alvo. Quando este bit é igual a zero significa que não está pronto para a transmissão.
3	RRDY	O sinalizador <i>Receive-Data-Ready</i> serve para indicar quando o microcontrolador está pronto para receber mensagens do transmissor. Quando este bit é igual a zero significa que não está pronto para a receção.
2	ARDY	O <i>Register-Access-Ready</i> é o sinalizador de que os registradores do módulo I2C estão prontos para serem acessados quando este bit é igual a um. Este bit não possui efeito quando o microcontrolador está trabalhando como o alvo.
1	NACK	<i>No-acknowledgement</i> bit serve para sinalizar se ocorreu um bit do tipo NACK depois dos dados. Quando o receptor envia um ACK, este bit é igual a zero e quando o receptor envia um NACK, este bit é igual a um.
0	AL	<i>Arbitration-Lost</i> quando este bit é igual a um, sinaliza a perda de arbitragão. Este bit não é relevante quando o microcontrolador é o alvo.

Adaptado de Texas Instruments (2017)

3.2.3.7 Registrador de habilitação das interrupções I2CIER

O registrador I2CIER é o responsável por habilitar e desabilitar as interrupções no microcontrolador. Possui dezesseis bits, mas somente sete bits possuem funcionalidades. A Tabela 8

Tabela 8 – Descrição dos bits do registrador de interrupções I2CIER

Bit	Nome	Descrição
15-7	Reservado	Bits reservados que não possuem efeito no funcionamento.
6	AAS	<i>Addressed As Slave</i> bit, quando o sexto bit do I2CIER é igual a um, uma interrupção irá acontecer quando o microcontrolador identificar seu endereço de alvo na rede.
5	SCD	<i>Stop Condition Detected</i> é o bit que habilita a interrupção causada pela detecção da requisição de parada na linha SDA.

Continua...

Bit	Nome	Descrição
4	XRDY	A interrupção <i>Transmit-Data-Ready</i> serve para indicar quando o microcontrolador está pronto para transmitir mensagens para o receptor. Esta interrupção é habilitada com o bit XRDY é igual a um.
3	RRDY	O interrupção <i>Receive-Data-Ready</i> serve para indicar quando o microcontrolador está pronto para receber mensagens. Esta interrupção é habilitada com o bit RRDY é igual a um.
2	ARDY	A interrupção <i>Register-Access-Ready</i> está habilitada quando o bit ARDY do I2CIER é igual a um. Quando os registradores do módulo I2C estão prontos, o bit ARDY do I2CSTR é igual a um.
1	NACK	A transmissão do <i>No-acknowledgement</i> bit na linha SDA pode causar uma interrupção no microcontrolador quando este é igual a um.
0	AL	A interrupção <i>Arbitration-Lost</i> só é relevante quando o microcontrolador é um dos controladores na rede. Quando este bit é igual a um, quando o microcontrolador perder a arbitragem da linha SDA irá causar uma interrupção.

Fonte: Adaptado de Texas Instruments (2017)

4 DESENVOLVIMENTO

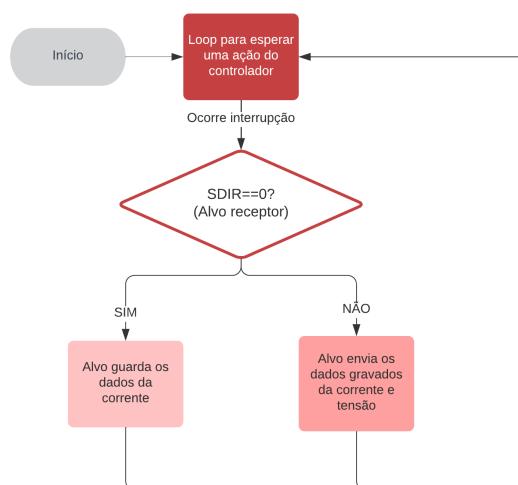
Este capítulo apresenta como as informações presentes no Capítulo 2 aplicam-se a este projeto. Aqui é proposto uma maneira de utilizar o RPi 4, como controlador, em conjunto com o microcontrolador Piccolo f28069, alvo, para realizar o controle da potência fornecida ao definir o valor de corrente. Os valores de tensão serão constantes, portanto, somente o valor de corrente será enviado do Raspberry para o microcontrolador, mas a leitura será feita da tensão e da corrente.

Neste projeto o RPi envia os valores de corrente com duas casas decimais, o valor de corrente é multiplicado por cem e dividido em dois bytes, constituído pelo *Most Significant Byte* (MSB), byte mais significativo, e pelo *Least Significant Byte* (LSB), byte menos significativo. O microcontrolador recebe o MSB e LSB, guarda esses valores nos registradores, envia a mesma corrente quando solicitado e envia os valores de tensão também dividido em dois bytes.

4.1 FUNCIONAMENTO DO MICROCONTROLADOR COMO O ALVO DO PROJETO

O microcontrolador Piccolo controlstick F28069 tem a função de servir como intermediador do RPi e o equipamento no qual deseja-se controlar a potência. Neste projeto ele foi utilizado somente para simular esta comunicação, recebendo os valores de corrente enviados pelo controlador, transmitindo os mesmos valores de corrente recebidos e enviando também um valor de tensão. O diagrama de fluxo mostrado na Figura 22 mostra o funcionamento do microcontrolador de maneira simplificada.

Figura 22 – Diagrama de funcionamento da comunicação com o microcontrolador.



Fonte: Autoria própria

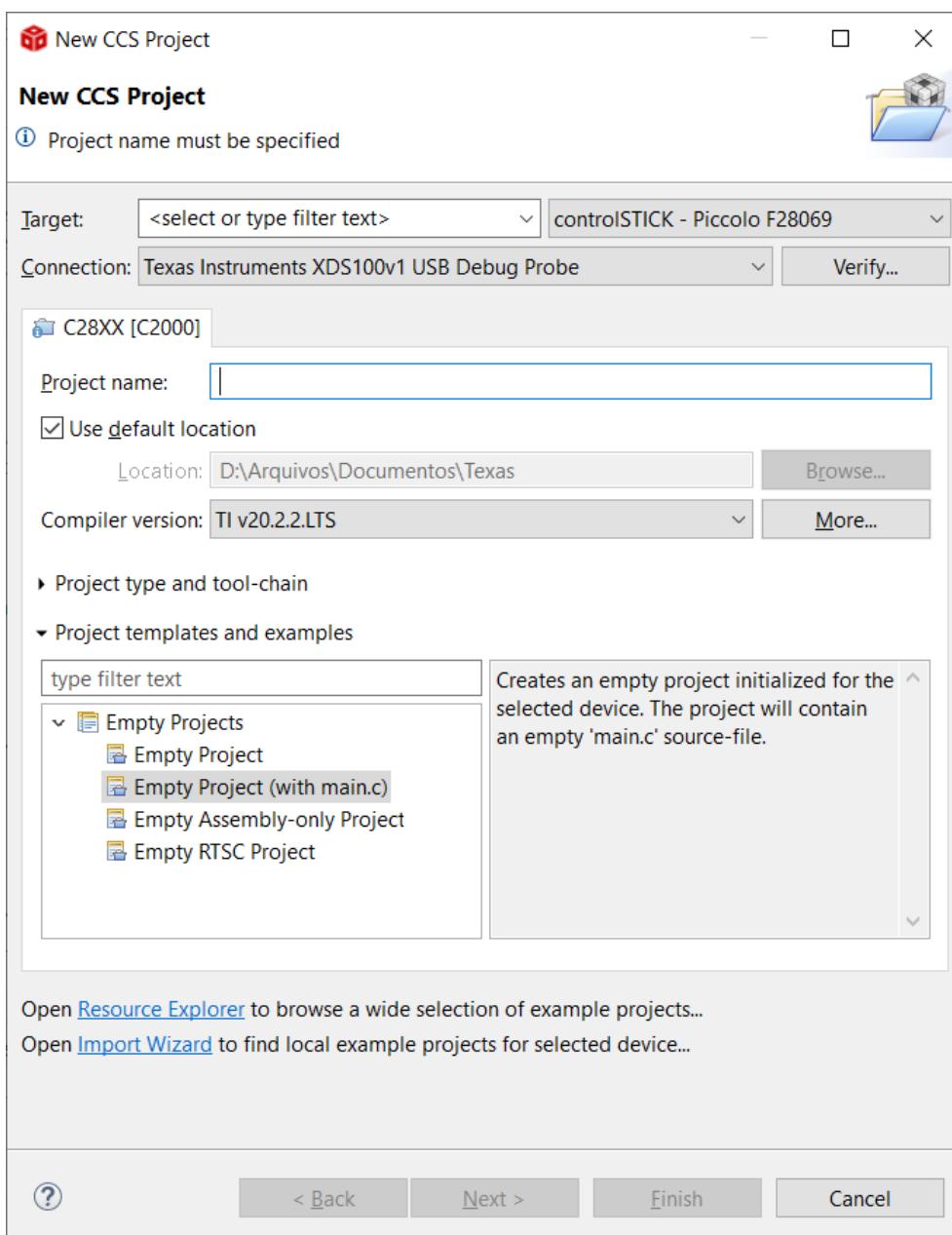
Na Figura 22 é possível notar que nada ocorre com o microcontrolador enquanto

o controlador não gerar uma interrupção. Vale notar também que após o envio ou recepção de uma mensagem, o alvo volta para estado de espera.

4.1.1 Configurações do microcontrolador

Como mencionado no Capítulo 2, o programa utilizado para programar o Piccolo controlstick F28069 é o CCS. Para iniciar um novo projeto é necessário escolher modelo correto do microcontrolador, selecionar um programa em branco, como mostra a Figura 23.

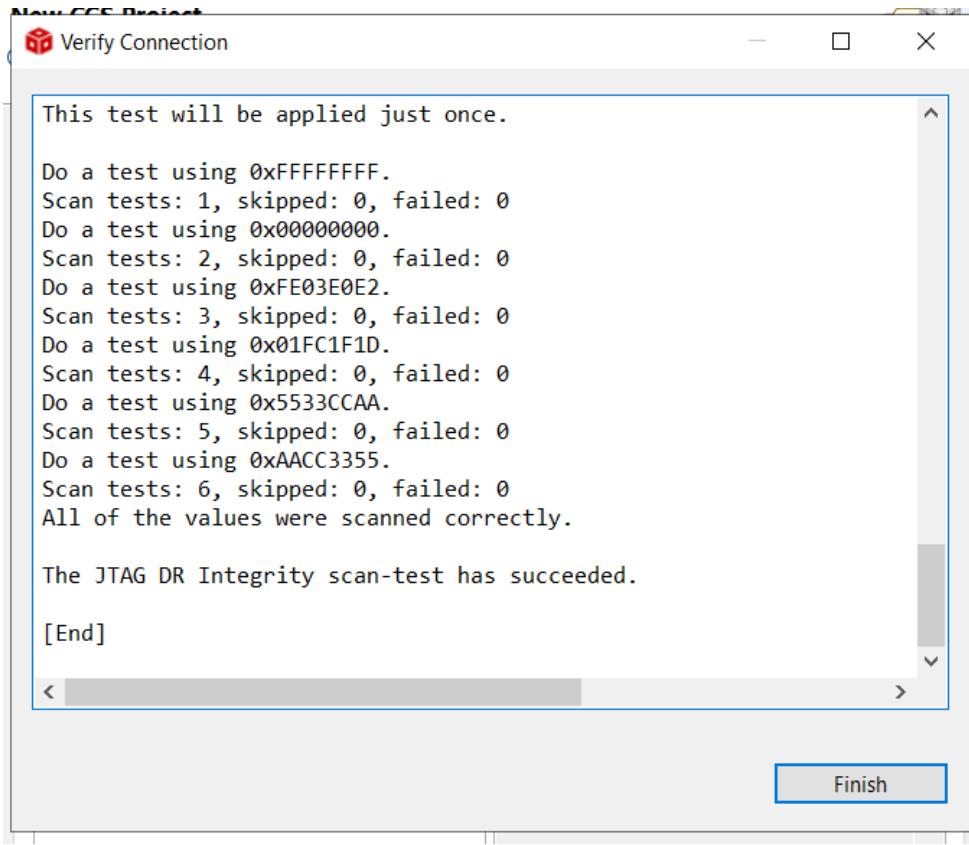
Figura 23 – Configuração inicial para criar um novo projeto.



Fonte: Autoria própria

Ao selecionar o microcontrolador desejado, neste projeto o *Target* é o controlS-TICK - Piccolo F28069 mostrado na Figura 23, e a conexão com uma conexão USB. Ao clicar no botão *Verify* é esperado que uma janela similar à Figura 24 apareça, confirmando que a conexão entre o computador e o microcontrolador foi estabelecida corretamente.

Figura 24 – Janela de confirmação da conexão entre microcontrolador e computador.



Fonte: Autoria própria

O C2000WARE versão 3.04 oferece vários arquivos para serem usados como base, neste caso os arquivos de interesse podem ser encontrados na pasta C:\ti\c2000\C2000Ware_3_04_00_00\device_support\f2806x\. Para este projeto foram importados alguns dos arquivos disponíveis dentro deste diretório, a Tabela 9 mostra os arquivos necessários e em quais sub-pastas estão localizados.

Tabela 9 – Arquivos necessários na importação do CCS

Nome do arquivo	Pasta
Example_2806xI2C_eeprom	... \examples\c28\i2c_eeprom
F2806x_Headers_nonBIOS	... \headers\cmd
F2806x_GlobalVariableDefs	... \headers\source
28069_RAM_Ink	... \common\cmd
F2806x_I2cDefines	... \common\include
F2806x_CodeStartBranch	... \common\source
F2806x_CpuTimers	... \common\source
F2806x_DefaultIlsr	... \common\source
F2806x_I2C	... \common\source
F2806x_PieCtrl	... \common\source
F2806x_PieVect	... \common\source
F2806x_SysCtrl	... \common\source
F2806x_usDelay	... \common\source

Fonte: Adaptado de [C2000]

4.1.2 Arquivos do projeto no microcontrolador

Dentre os arquivos importados existe o F2806x_I2C.c que define quais os pinos utilizados para a comunicação I2C. Por definição ele vem com os pinos 28 e 29 configurados como o SDA e SCL respectivamente, mas neste projeto foi optado pela utilização dos pinos 32 e 33. A mudança de pinos depende de habilitar o resistor de pull-up dos pinos desejados e também desabilitar o pull-up dos pinos 28 e 29. Para realizar esta mudança basta desabilitar as linhas de códigos adicionando duas barras (//) no lado esquerdo da linha e remover estas barras das linhas em que deseja habilitar, da mesma maneira mostrada na Figura 25.

Figura 25 – Alterações nos pinos de comunicação I2C do arquivo F2806x_I2C.c.

```

InitI2CGpio()
{
    EALLOW;
    // Enable internal pull-up for the selected pins
    //GpioCtrlRegs.GPAPUD.bit.GPIO28 = 0;      // Enable pull-up for GPIO28 (SDAA)
    //GpioCtrlRegs.GPAPUD.bit.GPIO29 = 0;      // Enable pull-up for GPIO29 (SCLA)
    GpioCtrlRegs.GPBPU0D.bit.GPIO32 = 0;      // Enable pull-up for GPIO32 (SDAA)
    GpioCtrlRegs.GPBPU0D.bit.GPIO33 = 0; // Enable pull-up for GPIO33 (SCLA)

    // Set qualification for selected pins to asynch only
    // This will select asynch (no qualification) for the selected pins.
    //GpioCtrlRegs.GPAQSEL2.bit.GPIO28 = 3; // Asynch input GPIO28 (SDAA)
    //GpioCtrlRegs.GPAQSEL2.bit.GPIO29 = 3; // Asynch input GPIO29 (SCLA)
    GpioCtrlRegs.GPBQSEL1.bit.GPIO32 = 3; // Asynch input GPIO32 (SDAA)
    GpioCtrlRegs.GPBQSEL1.bit.GPIO33 = 3; // Asynch input GPIO33 (SCLA)

    // This specifies which of the possible GPIO pins will be I2C functional pins.
    //GpioCtrlRegs.GPAMUX2.bit.GPIO28 = 2; // Configure GPIO28 for SDAA operation
    //GpioCtrlRegs.GPAMUX2.bit.GPIO29 = 2; // Configure GPIO29 for SCLA operation
    GpioCtrlRegs.GPBMUX1.bit.GPIO32 = 1; // Configure GPIO32 for SDAA operation
    GpioCtrlRegs.GPBMUX1.bit.GPIO33 = 1; // Configure GPIO33 for SCLA operation
    EDIS;
}

```

Fonte: Autoria própria

Ainda dentro da função InitI2CGPIO é necessário alterar o modo de funcionamento dos pinos para que trabalhem de maneira assíncrona, mostrado na parte central da Figura 25. A última alteração no arquivo é em relação aos pinos que serão utilizados como SDA e SCL, mostrado na parte inferior da Figura 25.

O arquivo contendo o funcionamento do microcontrolador pode ser encontrado no apêndice - A, trata de alterações realizadas no arquivo Example_2806xI2C_eeprom.c importado do C2000WARE.

4.1.3 Microcontrolador no modo alvo receptor

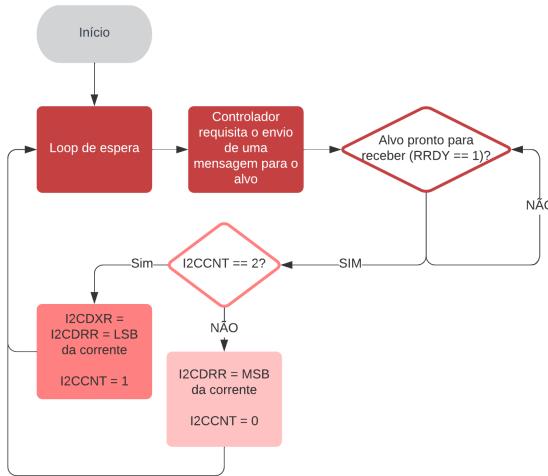
Quando o microcontrolador recebe uma interrupção do controlador para que receba os dados, o registrador SDIR é igual a zero. A flag RRDY indica se o alvo está preparado para receber os dados que o controlador quer enviar, caso não esteja preparado para a recepção, o RRDY é igual a zero e muda para um quando está pronto. O código que controla a recepção contorna este problema esperando que a flag seja igual a um, quando ocorre a alteração o registrador de interesse é o contador, I2CCNT. Caso o valor do contador seja igual a dois, significa que o alvo está recebendo a primeira parte do valor de corrente, o controlador está enviando o LSB. O valor é recebido pelo registrador de recepção, I2CDRR, este valor é então movido para

o registrador de transmissão, I2CDXR. Após o LSB ser movido de registradores, o registrador de contagem recebe o valor um.

Com o I2CCNT igual a um, o próximo valor enviado pelo controlador será o MSB e ficará guardado no próprio registrador I2CDRR, o registrador do contador irá ser alterado de um para zero.

Estas etapas são demonstradas no diagrama de fluxo da Figura 26.

Figura 26 – Diagrama de fluxo da recepção de dados pelo microcontrolador.



Fonte: Autoria própria

Ao final da recepção do valor de corrente o I2CCNT é igual a zero, o LSB está guardado no I2CDXR que é o registrador de envio e o MSB não é movido, ou seja, fica guardado no I2CDRR que é o registrador de recepção de dados.

4.1.4 Microcontrolador no modo alvo transmissor

Quando o microcontrolador recebe uma interrupção do controlador para que envie os dados, o registrador SDIR é igual a um, mas para que o envio se inicie a flag XRDY precisa ser igual a um, enquanto ela for igual a zero, os dados não serão enviados. A Figura 27 mostra esta espera do microcontrolador para poder transmitir.

Se o valor do I2CCNT for igual a zero, será enviado o valor do I2CDXR que corresponde ao LSB do valor da última corrente enviada, depois da transmissão o I2CCNT é incrementado em um.

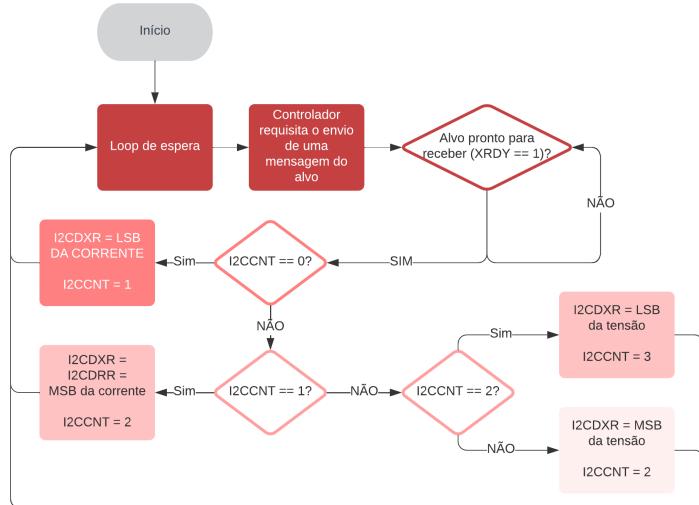
Se o valor do I2CCNT for igual a um, será enviado o MSB da corrente que está escrito no I2CDRR, este valor é movido para o I2CDXR e transmitido para o controlador. Após a transmissão da corrente, espera-se que o controlador requisite a tensão, o alvo se prepara para a requisição ao incrementar o valor registrado no I2CCNT, totalizando o valor dois.

Depois de transmitir os bytes da corrente, os próximos valores a serem transmitidos serão o MSB e LSB da tensão. Para isto foi escolhido o valor 10,10 volts para representar a tensão. Antes de transmitir o valor de tensão medida, o valor é multiplicado por cem, resultando no número mil e dez da base decimal, convertido para a base hexadecimal é 0x03F2, portanto o MSB é 0x03 e o LSB é 0xF2.

O controlador gera uma nova interrupção, requisitando o envio do valor de tensão para o alvo, quando esta interrupção ocorre o I2CCNT é igual a dois e envia o LSB da tensão para o controlador. A contagem de bytes enviados é atualizada para três.

O último byte a ser enviado é o MSB da tensão, neste caso o controlador causa uma nova interrupção, por conta do I2CCNT estar registrado com o valor três, sendo enviado o MSB da tensão e o I2CCNT volta para o valor dois, indicando que está pronto para receber os dois bytes da corrente.

Figura 27 – Diagrama de fluxo da transmissão de dados pelo microcontrolador.



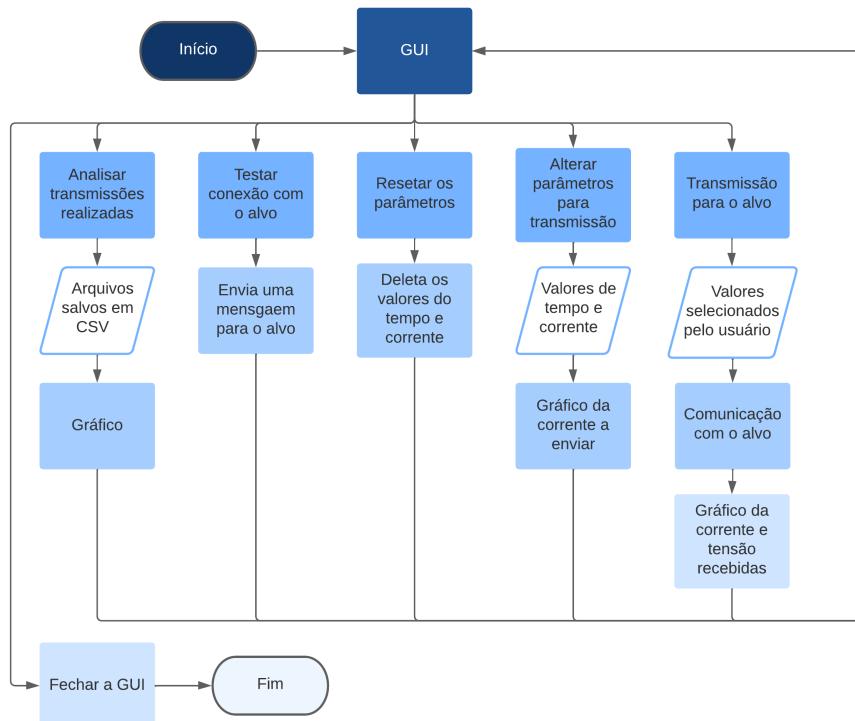
Fonte: Autoria própria

Em resumo, o I2CCNT inicia com o valor zero, envia o LSB da corrente guardado no I2CDXR e aumenta o I2CCNT para um. Quando o controlador requisita o envio de uma nova mensagem é enviado o MSB da corrente ao mover o valor salvo no I2CDRR para o I2CDXR e incrementa o I2CCNT em um, totalizando dois. Depois de enviar a corrente, o próximo valor que será enviado é o LSB da tensão e o I2CCNT será igual a três. O último valor a ser enviado será o MSB da tensão, o I2CCNT será igual a dois após esta transmissão.

4.2 FUNCIONAMENTO DO RASPBERRY PI COMO CONTROLADOR DO PROJETO

O RPi é o responsável pelo controle da comunicação. Para que a comunicação ocorra dentro do esperado existem vários passos para confirmar que os parâmetros estão dentro do desejado. A Figura 28 mostra o diagrama de fluxo com as principais funções desempenhadas.

Figura 28 – Diagrama de fluxo das funções do RPi.



Fonte: Autoria própria

4.2.1 Configuração dos parâmetros

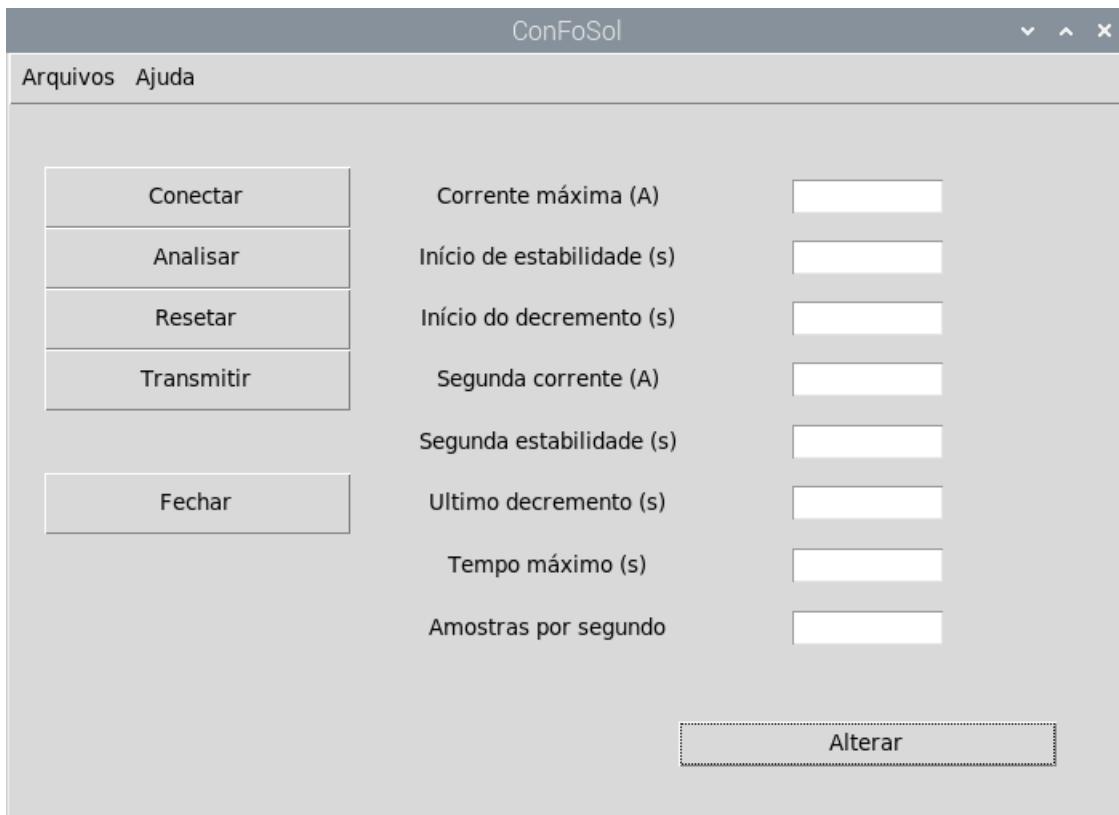
Os parâmetros que necessitam de definição por parte do usuário são:

- Corrente máxima da estabilidade em amperes.
- Corrente menor da estabilidade em amperes.
- Tempo de aumento da corrente em segundos.
- Tempo de estabilidade da corrente máxima em segundos.
- Tempo de decréscimo da corrente máxima até a menor em segundos.
- Tempo de estabilidade da corrente menor em segundos.

- Tempo de decréscimo da corrente menor até zero em segundos.
- Número de amostras por segundo.

A Figura 29 mostra as caixas de texto do programa no qual os valores citados acima devem ser inseridos.

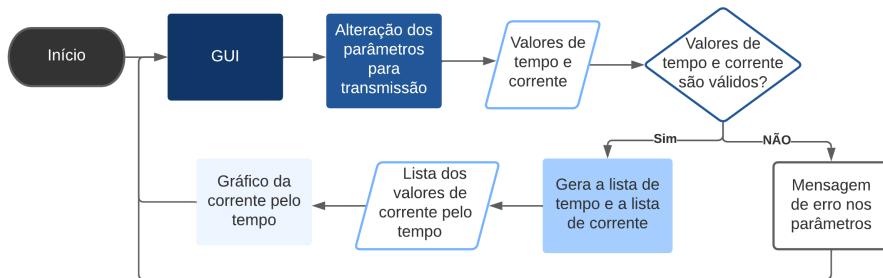
Figura 29 – Janela inicial do programa com as caixas de valores em branco.



Fonte: Autoria própria

Caso os valores estabelecidos pelo usuário sejam válidos, é gerada uma lista contendo os valores de tempo em que a corrente deve ser transmitida. Esta lista é utilizada para criar dos valores que a corrente deve alcançar de acordo com o tempo decorrido. Com as listas de tempo e corrente determinadas, um gráfico da corrente em função do tempo é gerado, o diagrama de fluxo da Figura 30 mostra a sequência das etapas.

Figura 30 – Diagrama de fluxo da alteração dos parâmetros pelo usuário.



Fonte: Autoria própria

4.2.1.1 Lista de tempo

Os valores de corrente são gerados a partir do número de amostras desejado pelo usuário. A Equação 2 mostra a relação entre $Amostras_{/s}$, número de amostras por segundo, e $T_{Amostra}$, período de tempo entre as amostras.

$$T_{Amostra} = \frac{1}{Amostras_{/s}} \quad (2)$$

Os itens dentro da lista de tempo possuem quatro casas decimais, ou seja, quando o usuário seleciona oito amostras por segundo, utilizando a Equação 2, temos que a cada 0,125 segundos, o controlador irá enviar um novo valor de corrente para o alvo. A lista de tempo sempre inicia em zero, o primeiro valor adicionado é o $T_{Amostra}$, o segundo é igual ao valor do primeiro mais o $T_{Amostra}$ e a cada nova iteração é adicionado o $T_{Amostra}$ ao item anterior da lista até que alcance o limite de tempo determinado pelo usuário.

Para cada amostra desejada pelo usuário é necessário o envio do valor da corrente, recebido o valor da corrente medida e da tensão, cada valor precisa de dois bytes para ser transmitido, então para cada amostra são transferidos seis bytes de dados.

O RPi e o microcontrolador tem por padrão a velocidade de cem quilo hertz por segundo para transmissão no protocolo I2C. As mensagens entre eles são constituídas por um bit de início, sete bits de endereço do alvo, um bit de direcionamento da linha SDA, um bit de ACK do alvo, oito bits de dados, um bit de ACK do receptor e um bit de parada, totalizando vinte bits. A cada ciclo de clock na SCL é possível enviar um bit na SDA, com cem quilo bits por segundo é possível transferir cinco mil mensagens de vinte bits.

Para a transferência de seis bytes de dados são necessárias seis mensagens, levando em consideração que cada amostra possui seis bytes, a transferência selecio-

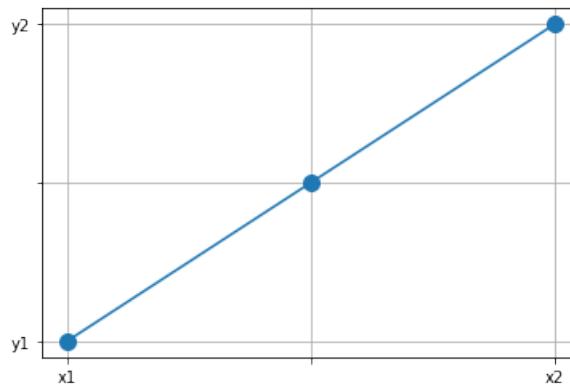
nada pelo usuário é limitada em oitocentas e trinta e três amostras por segundo.

4.2.1.2 Lista dos valores de corrente

O primeiro passo para a comunicação é definir a função da corrente pelo tempo, a corrente pode ser separada em cinco retas diferentes. A primeira reta é crescente e vai até o valor máximo de corrente, a segunda é constante, a terceira é decrescente, a quarta é constante e a quinta é uma reta decrescente.

A equação de uma reta pode ser encontrada a partir de dois pontos, utilizando os pontos $A(x_1, y_1)$ e $B(x_2, y_2)$, é possível encontrar um terceiro ponto $C(x, y)$ na mesma reta como mostra a Figura 31.

Figura 31 – Reta do ponto A até o ponto B.



Fonte: Autoria própria

A Equação 3 utiliza o método o conceito da determinante da matriz para encontrar o ponto C (ROSSI, 2019).

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x & y & 1 \end{vmatrix} = 0 \quad (3)$$

Ao aplicar a determinante na Equação 4 e isolar o y , obtemos a Equação 4, equação das retas.

$$y = \frac{x(y_1 - y_2) + x_1 \cdot y_2 - x_2 \cdot y_1}{x_1 - x_2} \quad (4)$$

A primeira reta representa o único aumento de corrente, os pontos de interesse são o valor máximo de corrente i_{max} e o momento em que a corrente para de aumentar t_1 , como a reta inicia no tempo e corrente zero, a reta de interesse vai do ponto $A(0, 0)$

até o ponto $B(t_1, i_{max})$, a Equação 5 é obtida ao inserir os pontos $A(0, 0)$ e $B(t_1, i_{max})$ na Equação 4.

$$y = \frac{x \cdot i_{max}}{t_1} \quad (5)$$

Com a Equação 5 é possível encontrar qualquer ponto dentro de um reta crescente que passa pelo ponto $A(0, 0)$.

A segunda e quarta reta são constantes, em qualquer instante de tempo o valor de corrente será o mesmo. No caso da segunda reta será sempre o valor de corrente máxima, já na quarta reta será o valor de corrente menor.

A terceira e a quinta reta são retas decrescentes, onde o valor de corrente do ponto $A(x_1, y_1)$ será sempre maior do que a corrente do ponto $B(x_2, y_2)$ e são definidas pela Equação 4.

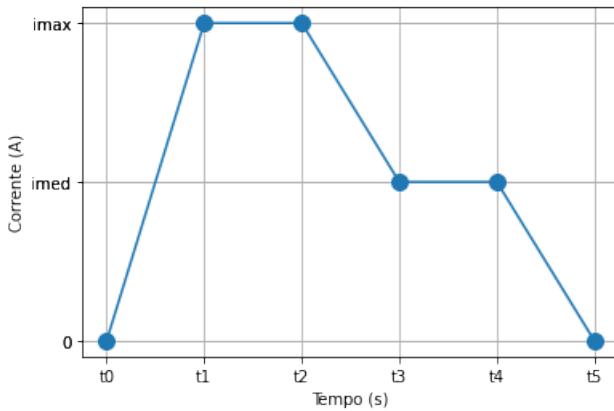
4.2.1.3 Gráfico da corrente transmitida

O gráfico da corrente é feito utilizando a biblioteca matplotlib do Python. A abscissa é a própria lista de tempo e a ordenada corresponde a lista das correntes.

Com a biblioteca do matplotlib instalada no RPi, basta utilizar o comando `import matplotlib` no início do arquivo em Python para utilizar as funções disponíveis desta biblioteca.

Ao traçar as retas da corrente pelo tempo, o resulta será similar ao mostrado na Figura 32.

Figura 32 – Gráfico da corrente transmitida pelo tempo.



Fonte: Autoria própria

4.2.2 Reset dos parâmetros

Para limpar as caixas de texto dos parâmetros basta clicar no botão Resetar, o resultado é mostrado na Figura 29.

4.2.3 Teste de conexão com o alvo

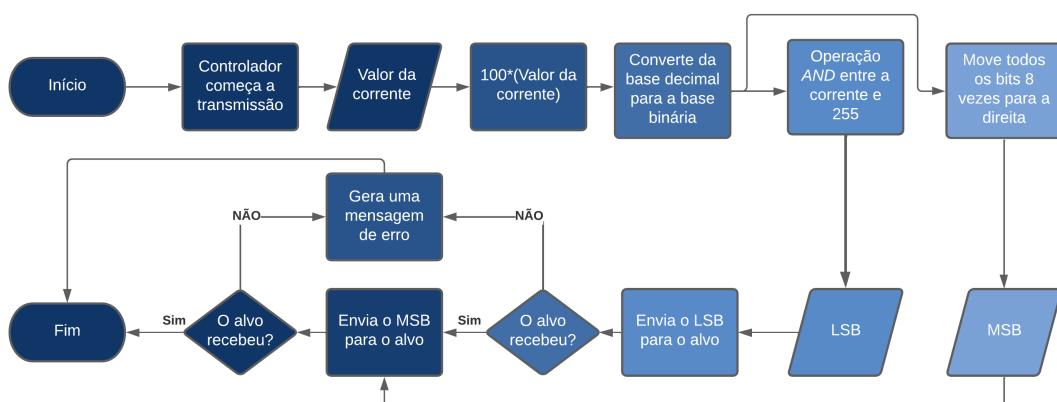
Para verificar se o alvo está conectado com o controlador no endereço desejado é necessário enviar uma mensagem. A biblioteca SMBus é extremamente útil, para usá-la é preciso importar a biblioteca no projeto ao utilizar o comando `import SMBus` no início do arquivo.

Um dos comandos disponíveis na biblioteca do SMBus é `self.bus.write_quick` (endereço do alvo) que envia uma mensagem para o endereço desejado, caso o alvo não receba a mensagem, um erro é gerado.

4.2.4 Transmissão de corrente

Com o alvo conectado com o controlador e com as listas de tempo e corrente geradas, é possível iniciar a comunicação. A transmissão utiliza a biblioteca do SMBus, o comando `self.bus.write_byte`(endereço, dados) faz o envio de dados contendo oito bits do controlador para o alvo, mas como os valores de corrente possuem dezesseis bits, são necessários dois envios para que todo o valor seja entregue. O diagrama da Figura 33 mostra o processo de transmissão.

Figura 33 – Diagrama de fluxo da corrente enviada para o alvo.



Fonte: Autoria própria

A corrente com duas casas decimais é multiplicada por cem, esse valor que agora é um número inteiro, é convertido da base decimal para a base binária. Na base binária duas operações são realizadas, a primeira é a mudança dos bits da corrente em oito vezes para a direita, por exemplo o número binário 1001 1001 0110 0110 quando movido oito vezes para a direita é igual a 0000 0000 1001 1001, ou seja, é o byte mais significativo. O byte menos significativo pode ser utilizado ao realizar a operação booleana *AND* com o bit um, pois 1 *AND* 1 resulta em 1 e 0 *AND* 1 resulta

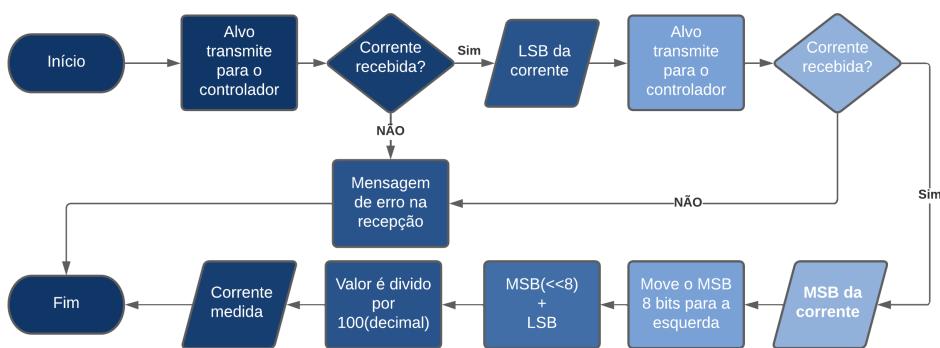
em zero. O resultado do número em base binária 1111 1111 AND qualquer número resultará nos oito primeiros bits.

Com os valores de MSB e LSB da corrente o controlador solicita o envio do LSB, se o alvo não receber causará uma mensagem de erro, mas se a recepção ocorrer fará o controlador enviar o MSB.

4.2.5 Recepção de corrente

Após a transmissão de corrente, o controlador solicita ao alvo a transmissão do valor de corrente medida. O diagrama de fluxo mostrado na Figura 34 mostra as etapas da comunicação.

Figura 34 – Diagrama de fluxo da corrente recebida pelo o controlador.



Fonte: Autoria própria

O controlador solicita o envio de mensagem do alvo, o alvo envia primeiramente o LSB da corrente que está guardado no registrador de transmissão. Após o recebimento do LSB por parte controlador, ele envia uma requisição de mensagem e o alvo envia o MSB da corrente. Depois da recepção dos bytes, o controlador faz o deslocamento de oito bits do MSB e soma a este valor o LSB, mas após esta soma de bytes, o valor ainda precisa ser dividido por cem para obter as duas casas decimais.

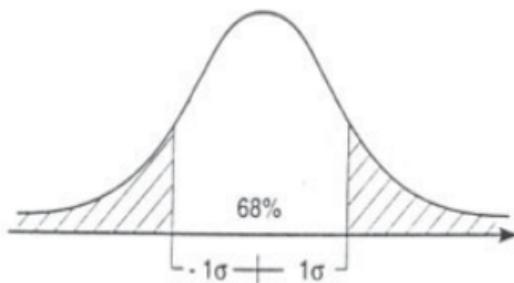
Este projeto somente simula a leitura da corrente, o alvo retransmite o valor de corrente enviado pelo controlador. Para uma boa visualização da corrente recebida é interessante que ela seja ligeiramente diferente da corrente transmitida, obtendo um resultado mais próximo da realidade.

Para tornar a visualização gráfica mais interessante o resultado da soma do MSB deslocado e LSB não é dividido por cem, mas por mil. Após a divisão ocorre a multiplicação com uma variável aleatória que fica entre nove e onze, em torno de setenta por cento das vezes, variável esta que segue a curva normal.

4.2.5.1 Distribuição normal

A distribuição normal é uma das distribuições contínuas mais importante dos estudos estatísticos, o seu gráfico é conhecido como curva normal ou curva gaussiana. No século dezoito ela ganhou notoriedade com os trabalhos de Abraham de Moivre, Pierre Simon Laplace e Carl Friedrich Gauss. A sua importância se deve ao fato da sua aparição em várias áreas de estudo, quando várias amostras são coletadas, muitas vezes assemelham-se com a curva normal mostrada na Figura 35 (SOUZA, 2019).

Figura 35 – Exemplo de curva gaussiana.



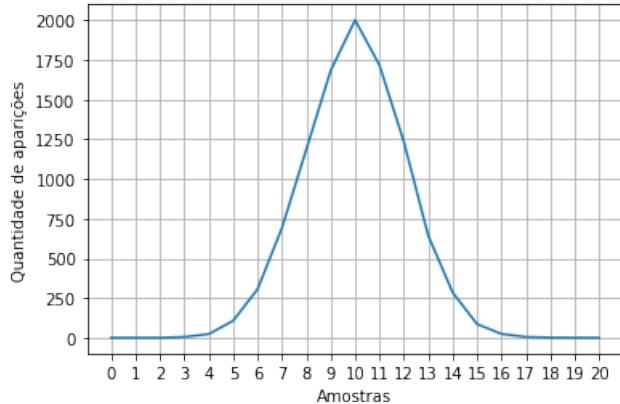
$$P(\mu-\sigma < X < \mu+\sigma) \approx 0,683$$

Fonte: Sousa (2019)

A Figura 35 mostra duas variáveis, uma delas é μ que representa a média, número que possui mais chance de aparecer, e a outra é σ , que representa a variância de dispersão da curva. Quanto maior for o σ , maior será a dispersão da curva.

Para a multiplicação da corrente foi escolhido um μ igual a dez, para que não seja necessário operar com números negativos, problema que apareceria ao utilizar um μ igual a um. O σ escolhido foi igual a um para que a dispersão da curva fosse pequena. A Figura 36 mostra o número de vezes que cada número apareceu de uma simulação com dez mil amostras geradas utilizando a biblioteca `random`.

Figura 36 – Exemplo de dez mil amostras geradas utilizando a distribuição normal.



Fonte: Sousa (2019)

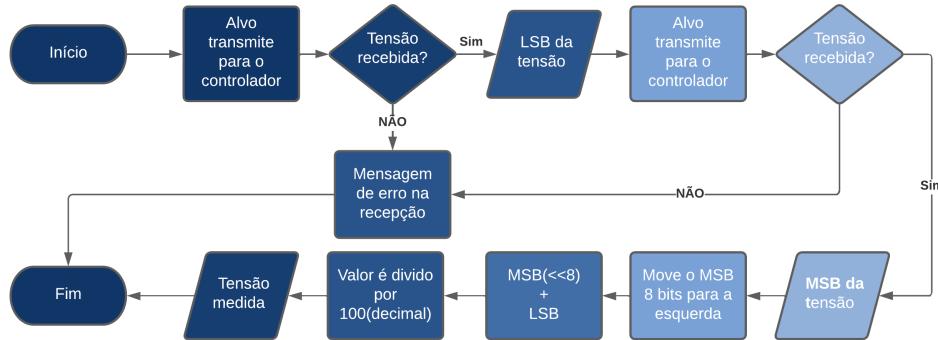
O comando `gauss(10, 1)` não gerou os números zero, um, dois, dezoito, dezenove e vinte em dez mil amostras.

4.2.6 Recepção de tensão

A recepção da tensão é muito similar à recepção da corrente, o controlador faz a requisição do envio da tensão e o alvo envia o LSB da tensão, para esta aplicação o LSB da tensão sempre será igual a 0xF2 em base hexadecimal. Após o controlador receber o LSB ele pede o envio do MSB da tensão que sempre será igual a 0x03 em base hexadecimal.

Assim como na recepção da corrente, o MSB da tensão é deslocado em oito bits para a esquerda, passando de 0x03 para 0x0300 em base hexadecimal, depois é somado com o LSB que resulta em 0x03F2 ou 1010 em base decimal. O valor então é dividido por cem resultando no valor de tensão igual a 10,10 na base decimal. Este processo é mostrado no diagrama de fluxo da Figura 37.

Figura 37 – Diagrama de fluxo da tensão recebida pelo o controlador.



Fonte: Autoria própria

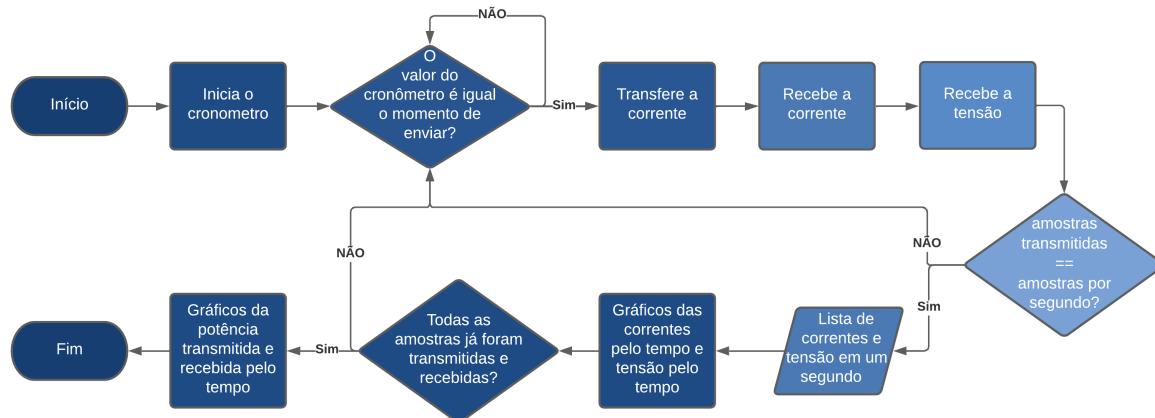
Neste projeto o controle da potência é feito exclusivamente pela corrente, não existe o envio de valores de tensão, então não há necessidade de multiplicar o valor da tensão recebida por uma variável.

4.2.7 Gráfico em tempo real da comunicação

O gráfico em tempo real é regido principalmente pela lista de tempo gerada para a transmissão, pois como mostra o diagrama de fluxo da Figura 38, a transmissão só ocorre se o tempo que o cronômetro está marcando é maior ou igual ao momento em que o controlador tem que transmitir para o alvo. Após a transmissão da corrente, vem a recepção da corrente medida e seguida pela tensão medida, estes valores são alocados em suas respectivas listas relacionadas com a lista de tempo.

Esta lógica na comunicação apresenta atrasos quando muitas amostras são transmitidas por conta da taxa de atualização do gráfico. Para contornar este problema, foi optado por atualizar o gráfico a cada um segundo, mas a comunicação ocorre de acordo com a lista de tempo, o gráfico é atualizado após o número de amostras transmitidas igualar-se com o número de amostras que devem ser enviadas em um segundo.

Figura 38 – Diagrama de fluxo de atualização do gráfico da comunicação.



Fonte: Autoria própria

Este processo se repete até que todas as amostras de corrente sejam transmitidas, medidas e recebidas. Quando não existem mais amostras para serem transmitidas, o RPi apresenta um novo gráfico comparando a potência esperada e potência real. A potência pode ser calculada utilizando a Equação 6.

$$P(t) = V(t) * i(t) \quad (6)$$

A Equação 6 é a equação da potência real instantânea, ela é a única utilizada pois as medidas são referentes ao secundário do circuito, onde a corrente não é alternada, somente contínua. Como as medições são realizadas de tempo em tempo, ou seja, a tensão V em volts e a corrente i em amperes são feitas quase ao mesmo tempo, então a potência instantânea obtida será extremamente próxima à potência real, com apenas micro segundos de diferença, dependendo do número de amostras por segundo.

4.2.8 Análise de arquivos

Depois de encerrar a comunicação é possível salvar os dados obtidos em arquivos no formato CSV, utilizando vírgulas como separador de dados. São salvos a lista de tempo, corrente transmitida, corrente recebida e tensão recebida, por padrão o nome do arquivo é igual aos parâmetros definidos pelo usuário, mas é possível alterar o nome do arquivo desde continue com a extensão .csv.

Dentro do programa é possível analisar os arquivos salvos em CSV, basta importá-los e os gráficos das correntes pelo tempo, tensão pelo tempo e potências pelo tempo serão gerados automaticamente.

4.3 CONEXÃO ENTRE O RASPBERRY PI E O MICROCONTROLADOR

Para que o RPi reconheça o microcontrolador é preciso habilitar o módulo I2C do RPi, o comando `sudo raspi-config` abre as configurações do software onde o módulo I2C pode ser encontrado em **Interfacing Options > Advanced Options > I2C > YES**.

Para atualizar os arquivos do sistema, as seguintes linhas de código são utilizadas na janela de comandos:

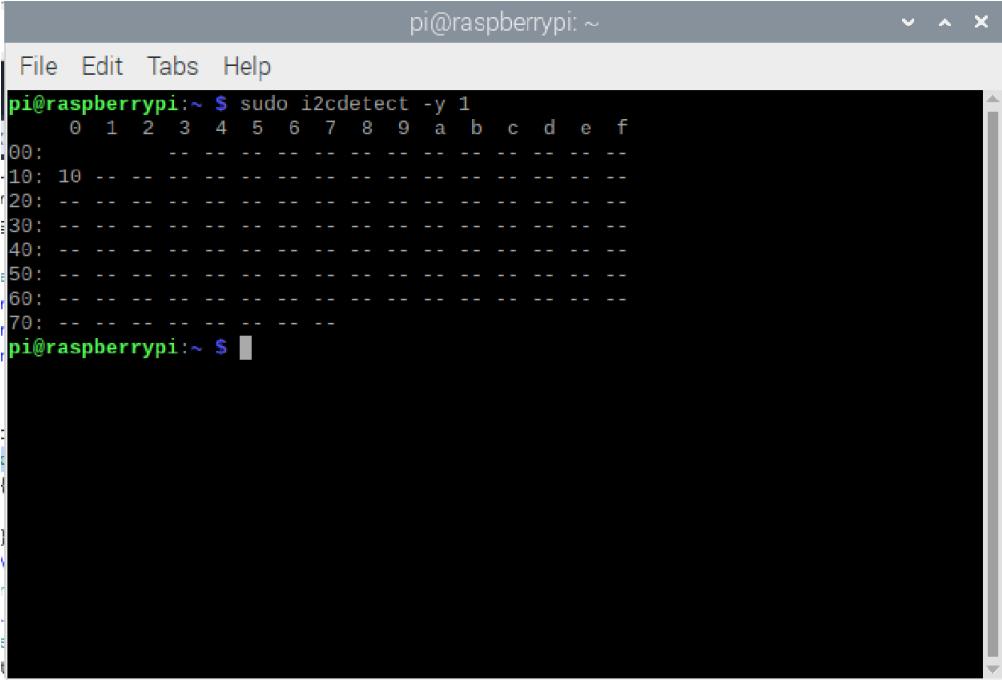
- `sudo apt-get update`
- `sudo apt-get upgrade`
- `sudo apt-get dist-upgrade`

Os passos seguintes são instalar a biblioteca `smbus` mencionada no Capítulo 2, o Python Dev para alterar configurações disponíveis para desenvolvedores, e ferramentas do módulo I2C com os seguintes comandos:

- `sudo apt-get install python-smbus python3-smbus`
- `sudo apt-get install python-dev python3-dev`
- `sudo apt-get install i2c-tools`

O pino 3 do RPi é o SDA do módulo I2C que deve ser conectado com o pino 13 do controlSTICK e linha SCL deve ser conectada entre o pino 5 do RPi e o pino 9 do controlSTICK. Para assegurar que a tensão nas linhas de SDA e SCL sejam as mesmas, o pino 9 do RPi e o pino 28 do controlSTICK devem ser conectados, igualando a tensão de aterramento. Caso o microcontrolador e o RPi estejam conectados corretamente, ao utilizar o comando `sudo i2cdetect -y 1` na janela de comandos, é esperado que apareça uma janela parecida com a Figura 39, indicando qual o endereço utilizado pelo alvo. Neste projeto o alvo ocupa o endereço 0x10 ou dezesseis em base decimal.

Figura 39 – Detecção de endereço do alvo.



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window has a menu bar with "File", "Edit", "Tabs", and "Help". The main area displays the output of the command "sudo i2cdetect -y 1". The output shows a 16x16 grid of characters representing I2C addresses. The columns are labeled 0 through f at the top, and the rows are labeled 00 through 70 on the left. Most cells contain a dash (-), indicating no device is present at that address. A few cells have other characters, such as '10' at row 10, column 10, and '20' at row 20, column 20.

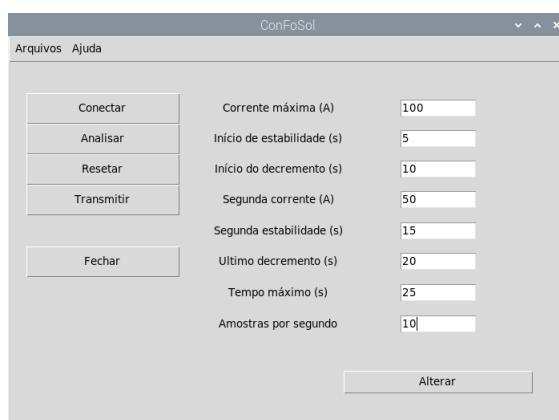
```
pi@raspberrypi:~ $ sudo i2cdetect -y 1
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: --
10: 10 --
20: 20 --
30: --
40: --
50: --
60: --
70: --
pi@raspberrypi:~ $
```

Fonte: Autoria própria

5 SIMULAÇÕES

O RPi utiliza o programa Thonny, que vem instalado no sistema operacional, para rodar o programa mostrado no apêndice B. Para exemplificar o funcionamento da comunicação foram utilizados os valores mostrados na Figura 40.

Figura 40 – Parâmetros para iniciar a transmissão.

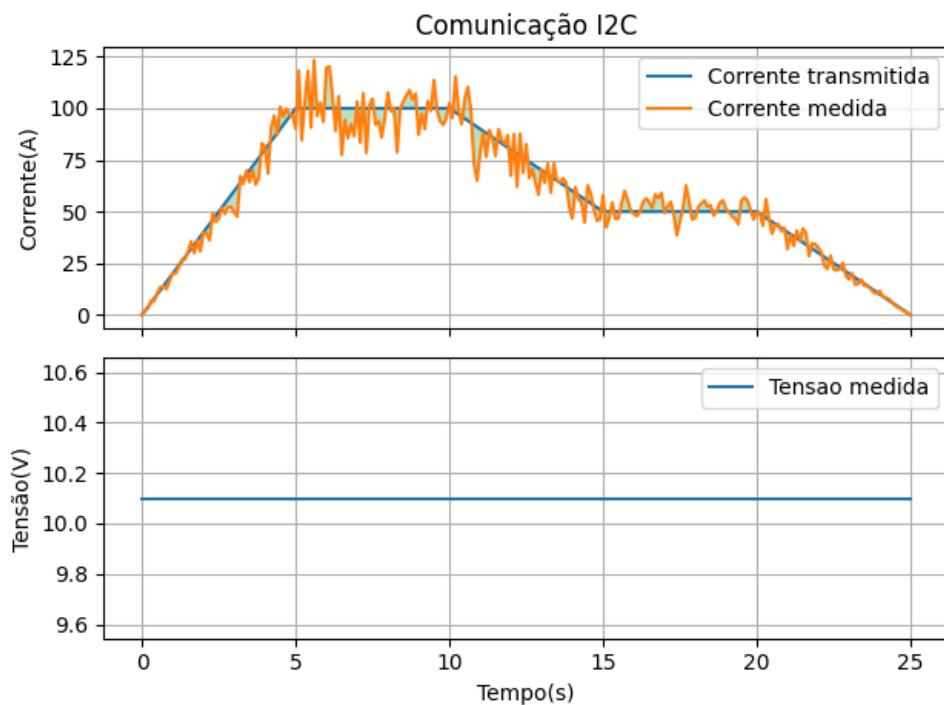


Fonte: Autoria própria

Ao clicar no botão Alterar, o gráfico da Figura 1 é gerado e representa a curva da corrente definida pelo usuário que será transmitida pelo controlador.

Caso o gráfico da Figura ?? esteja de acordo com o desejado, a comunicação irá iniciar com o clique no botão Transmitir. Durante a comunicação somente os gráficos das correntes e tensão serão exibidos, similar ao gráfico mostrado na Figura 41.

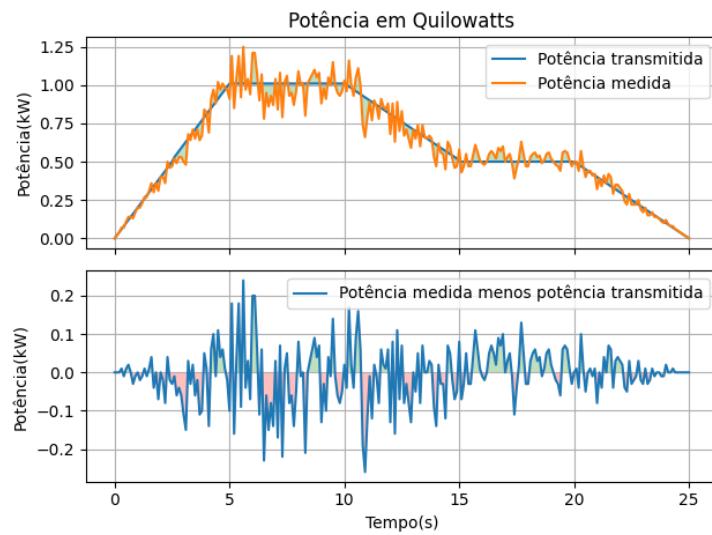
Figura 41 – Gráfico da corrente transmitida, corrente medida e tensão medida pelo tempo.



Fonte: Autoria própria

Quando todos os elementos da lista de valores de corrente são enviados e as listas de corrente e tensão recebidas possuem o mesmo número de elementos, um novo gráfico é gerado. Uma nova janela é apresentada com dois gráficos, o primeiro com o gráfico da potência enviada e da potência recebida e no segundo gráfico é apresentada a diferença entre a potência medida menos a potência transmitida, para facilitar a visualização da diferença entre a potência real medida da esperada.

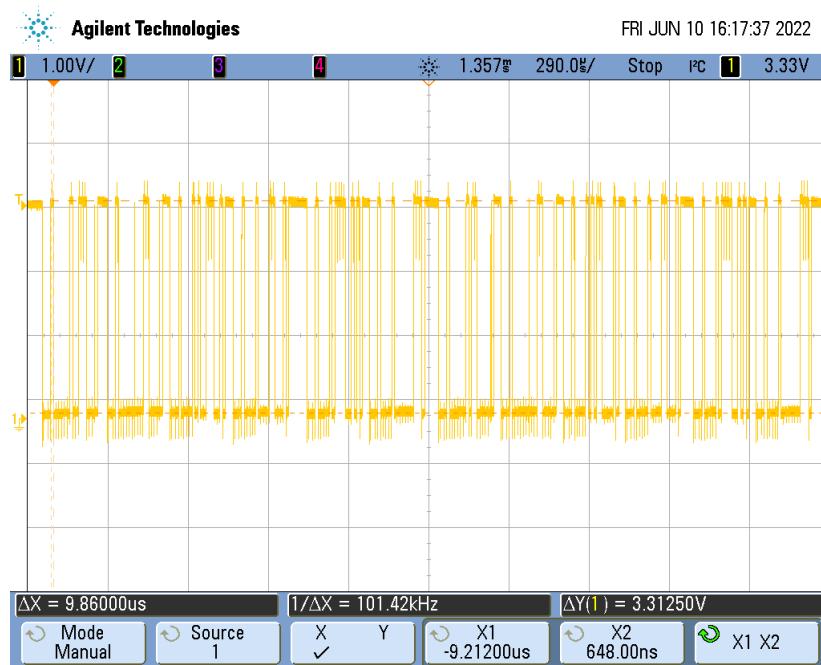
Figura 42 – Gráfico das potências e diferença entre a potência medida e a transmitida.



Fonte: Autoria própria

A Figura 43 mostra a forma de onda da linha SDA obtida na comunicação entre o RPI e o microcontrolador.

Figura 43 – Forma de onda de dados da linha SDA.

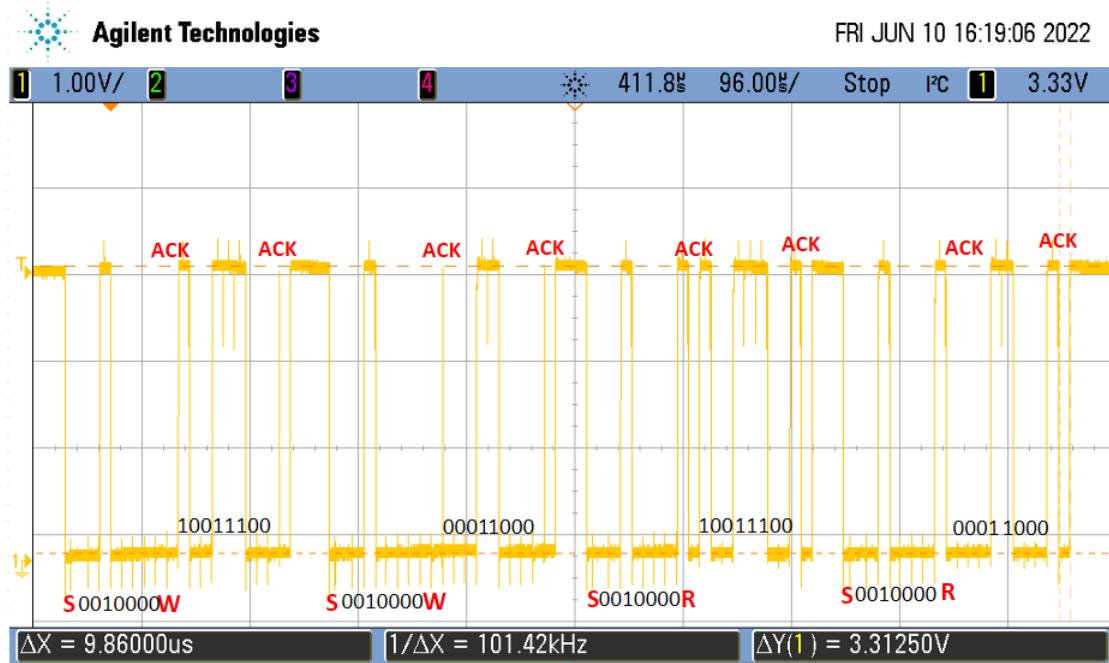


Fonte: Autoria própria

A Figura 43 possui muitos dados sendo exibidos, dificultando a visualização dos mesmos. A Figura 44 mostra o início da transmissão da Figura 43 em um espaço de

tempo menor, facilitando a identificação dos bits.

Figura 44 – Início da comunicação na linha SDA.



Fonte: Autoria própria

O microcontrolador possui o endereço 0x10 na base hexadecimal, na base binária esse endereço é igual a 0010000, ou dezesseis na base decimal. É possível notar que após o controlador sinalizar que vai iniciar a comunicação ao enviar o bit *S* é seguido pelo endereço do alvo e o bit de escrita(*W*) ou recebimento(*R*). Quando o controlador inicia a transmissão (*W*), o primeiro byte representa o LSB e o MSB é enviado na sequência. Na Figura 44 o LSB é 10011100 na base binária e o MSB é 00011000, juntando o MSB e LSB na base decimal o resultado é igual a 6300. Na segunda metade da Figura 44 o controlador faz a requisição do envio de dados por parte do alvo ao sinalizar o bit *R* após o endereço do alvo, onde o mesmo valor que foi enviado pelo controlador é então recebido pelo controlador representando a leitura feita pelo microcontrolador alvo.

Ao receber o valor 6300 na base decimal, o RPi faz a divisão deste número por 100, resultando no valor 63.00. Para que as linhas no gráfico não sejam exatamente iguais, o valor recebido é multiplicado por uma variável aleatória gaussiana.

6 CONCLUSÃO

Este trabalho teve como objetivo exemplificar a utilização do protocolo de comunicação para realizar a comunicação entre um controlador e um alvo, neste projeto o alvo não faz o controle de nenhum equipamento, mas com algumas alterações no microcontrolador será possível controlar uma fonte de solda.

Existem vários protocolos de comunicação, como visto no capítulo dois, neste projeto o protocolo I2C foi escolhido por oferecer a possibilidade de expansão do número de sensores utilizados. Além de um grande número de alvos controlados, esta comunicação possui uma boa taxa de transferência e um formato de mensagem que permite a detecção de erros através do bit ACK.

No capítulo três foram apresentados os programas utilizados para programar o RPi e microcontrolador, assim como arquivos adicionais e onde podem ser encontrados. As principais alterações foram comentadas, mas os códigos completos podem ser encontrados nos apêndices.

O quarto capítulo mostra a interface gráfica do projeto e as etapas para controlar a potência elétrica. Os gráficos da corrente e tensão são mostrados quase em tempo real, caso sejam mostrados a cada mensagem recebida ou enviada haverá um atraso na comunicação, portanto o gráfico é atualizado uma vez por segundo.

Para a utilização futura deste projeto, recomenda-se a utilização de poucas amostras por segundo, não mais que cem, pois apesar do gráfico ser atualizado uma vez por segundo, o número de amostras pode atrasar a visualização e por consequência atrasar a comunicação. Caso seja necessária a transmissão de várias amostras por segundo, é recomendado que o código em Python seja dividido em dois, um contendo o gráfico e outro separado realizando a comunicação e gravando os dados em um arquivo acessível para o gráfico.

REFERÊNCIAS

- COLTON, Shane. **TI Workshop + The Joys? of High-Speed Motor Control.** 2012. Disponível em: <http://scolton.blogspot.com/2012/04/ti-workshop-joys-of-high-speed-motor.html>. Acesso em: 8 ago. 2021.
- CORRIGAN, Steve. **Introduction to the Controller Area Network (CAN).** English. [S.I.], 2002.
- DHAKER, Piyu. **Introduction to SPI Interface.** 2018. Disponível em: <https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf>. Acesso em: 2 fev. 2022.
- ETIQ TECHNOLOGIES. **Basics of I2C communication.** 2016. Disponível em: <https://openlabpro.com/guide/basics-of-i2c/>. Acesso em: 8 ago. 2021.
- GHAEL, Hirak Dipak; SOLANKI, Dr. L; SAHU, Gaurav. A Review Paper on Raspberry Pi and its Applications. **International Journal of Advances in Engineering and Management (IJAEM),** 2020.
- LEENS, Frédéric. An Introduction to I2C and SPI Protocols. **IEEE Instrumentation Measurement Magazine,** 2009.
- MANKAR, Jayant *et al.* REVIEW OF I2C PROTOCOL. **International Journal of Research in Advent Technology,** 2014.
- MCKINNEY, Wes. **Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython.** [S.I.]: O'Reilly Media, 2017.
- MOTOROLA, Inc. **SPI Block Guide.** English. Versão 04.01. [S.I.], 2004.
- NXP SEMICONDUCTORS, Inc. **I2C-bus specification and user manual.** English. Versão 7.0. [S.I.], 2021.
- OUALLINE, Steve. **Practical C Programming.** [S.I.]: O'Reilly Media, 1997.
- PAZUL, Keith. Controller Area Network (CAN) Basics. **Microchip Technology Inc.,** 2002.
- PEÑA, Eric; LEGASPI, Mary. **UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter.** 2020. Disponível em: <https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html>. Acesso em: 8 ago. 2021.

RASPBERRY PI FOUNDATION. **Setting up your Raspberry Pi.** 2020. Disponível em: <https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up/0>. Acesso em: 8 ago. 2021.

RF WIRELESS WORLD. **Serial Transmission vs Parallel Transmission.** 2012. Disponível em: <https://www.rfwireless-world.com/Terminology/Serial-Transmission-vs-Parallel-Transmission.html>. Acesso em: 8 ago. 2021.

RITCHIE, Dennis M. The Development of the C Language. **Bell Labs/Lucent Technologies**, 1993.

ROSSI, Dênis Sidinei. **Estudando a equação geral da reta com o geogebra.** 2019. Universidade Federal de Santa Maria.

SOUSA, Áurea. O papel da distribuição normal na Estatística. **Correio dos Açores**, 2019.

TEXAS INSTRUMENTS, Inc. **TMS320F2806x Piccolo™ Microcontrollers.** English. [S.I.], 2018.

TEXAS INSTRUMENTS, Inc. **TMS320x2806x Piccolo Technical Reference Manual.** English. [S.I.], 2017.

YUVAYANA. **Definition, Classification of computer programming languages.** 2015. Disponível em: <https://er.yuvayana.org/definition-classification-of-computer-programming-languages/>. Acesso em: 2 fev. 2022.

APÊNDICE A – ARQUIVO EXAMPLE_2806XI2C_EEPROM.C DO C2000WARE COM AS ALTERAÇÕES

```

1 // ARQUIVO: Example_2806xi2c_eeprom.c
2 // TITULO: I2C EEPROM Example
3
4
5 // Included Files
6 #include "DSP28x_Project.h"      // Device Headerfile and
                                     Examples Include File
7
8
9 // Funcoes
10 void I2CA_Init(void);
11 __interrupt void i2c_int1a_isr(void);
12
13
14
15 //
16 // Main
17 //
18 void main(void)
19 {
20     // Primeiro Passo. Inicializar o Sistema de Controle:
21     // PLL, WatchDog, enable Peripheral Clocks
22     // O arquivo de exemplo e o arquivo F2806x_SysCtrl.c
23     //
24     InitSysCtrl();
25
26     // segundo passo. Inicializar o GPIO
27     // A funcao deste exemplo pode ser encontrada no arquivo
28     // F2806x_Gpio.c file e ilustra como preparar o GPIO
29     //
30     //InitGpio();
31
32     // Arquivo que seleciona os pinos GPIO especificamente para
33     // funcionar como I2C
34     //
35     InitI2CGpio();

```

```
36
37 // Terceiro Passo. Limpar todas as interrupcoes e
   // inicializar o PIE:
38 // Desabilitar interrupcoes CPU
39 //
40 DINT;
41
42 // Inicializa os registradores de controle PIE de forma
   // padrao
43 // Todas as flags desabilitadas
44 // Esta funcao e encontrada no arquivo F2806x_PieCtrl.c
45 //
46 InitPieCtrl();
47
48 // Desabilitar as interrupcoes CPU e limpar as flags
49 //
50 IER = 0x0000;
51 IFR = 0x0000;
52
53 //
54 // Inicializa os vetores do PIE com ponteiros para a
   // interrupcao do shell
55 // As rotinas do shell ISR podem ser encontradas em
   // F2806x_DefaultIsr.c.
56 // Esta funcao esta em F2806x_PieVect.c.
57 //
58 InitPieVectTable();
59
60 //
61 // Interrupts that are used in this example are re-mapped
   // to
62 // ISR functions found within this file.
63 //
64 EALLOW; // Isto e necessario para escrever nos
   // registradores protegidos do EALLOW
65 PieVectTable.I2CINT1A = &i2c_int1a_isr;
66 EDIS; // Isto e necessario para desabilitar a escrita nos
   // registradores protegidos do EALLOW
67
```

```
68      //  
69      // Quarto passo. Inicializar os perifericos do aparelho:  
70  
71      I2CA_Init();  
72  
73      // Quinto passo. Codigo do usuario  
74      //  
75  
76      // Habilita a interrupcao 1 do I2C no grupo 8 do PIE  
77      //  
78      PieCtrlRegs.PIEIER8.bit.INTx1 = 1;  
79  
80      // Habilita o CPU INT8 que esta conectado ao grupo 8 do PIE  
81      //  
82      IER |= M_INT8;  
83      EINT;  
84  
85      //  
86      // Loop de espera  
87      //  
88  
89      for(;;)  
90      {  
91  
92      }  
93  }  
94  
95 // Interrupcao vinda do I2C  
96 //  
97 void  
98 I2CA_Init(void)  
99 {  
100     //  
101     // Initialize I2C  
102     //  
103  
104  
105     I2caRegs.I2CPSC.all = 6;           // Prescaler - need 7-12  
          // Mhz on module clk
```

```
106     I2caRegs.I2CCLKL = 10;           // NOTE: must be non zero
107     I2caRegs.I2CCLKH = 5;           // NOTE: must be non zero
108
109     I2caRegs.I2COAR = 0x10;         // Own Address Register
110
111     I2caRegs.I2CIER.bit.AAS = 1;    // Addressed as slave
112         enabled
113     I2caRegs.I2CIER.bit.SCD = 1;    // Stop Condition Detected
114         enabled
115     I2caRegs.I2CIER.bit.XRDY = 1;   // Transmit Ready enabled
116     I2caRegs.I2CIER.bit.RRDY = 1;   // Receive Ready enabled
117     I2caRegs.I2CIER.bit.NACK = 1;   // No-Acknowledgment
118         enabled
119
120     I2caRegs.I2CMDR.bit.FREE = 1;
121
122     I2caRegs.I2CMDR.bit.MST = 0;    // Mode escravo
123     I2caRegs.I2CMDR.bit.TRX = 0;   // Modo receptor
124     I2caRegs.I2CMDR.bit.XA = 0;    // Modo de endereco 7 bits
125     I2caRegs.I2CMDR.bit.FDF = 0;   // Formato de dados livres
126         desabilitado
127     I2caRegs.I2CMDR.bit.BC = 0;    // 8 bits por byte de dado
128     I2caRegs.I2CMDR.bit.IRS = 1;   // Modulo I2C habilitado
129
130
131     I2caRegs.I2CCNT = 2;          // Utilizado para
132         transferir 16 bits
133
134     return;
135 }
136
137
138 // Fonte de interrupcoes do I2C
139 //
```

```
140     IntSource = I2caRegs.I2CISRC.all;
141
142     // Modulo recebe um sinal do mestre
143     if(IntSource == I2C_AAS_ISRC)
144     {
145         // Escravo receptor
146         if(I2caRegs.I2CSTR.bit.SDIR == 0)
147         {
148             // Espera o modulo estar pronto para receber
149             // informacoes
150             while(I2caRegs.I2CSTR.bit.RRDY == 0)
151             {
152                 }
153                 // Envia a primeira parte dos dados
154                 if(I2caRegs.I2CCNT == 2)
155                 {
156                     I2caRegs.I2CDXR = 1*I2caRegs.I2CDRR;
157                     I2caRegs.I2CCNT = 1;
158                 }
159                 // Se a primeira parte foi enviada, envia a segunda
160                 if(I2caRegs.I2CCNT == 1)
161                 {
162                     I2caRegs.I2CCNT = 0;
163                 }
164             }
165
166             // Escravo transmissor
167             else if(I2caRegs.I2CSTR.bit.SDIR == 1)
168             {
169                 // Espera o modulo estar pronto para enviar
170                 while(I2caRegs.I2CSTR.bit.XRDY == 0)
171                 {
172                     }
173                     // Se o contador for igual a zero entao pode
174                     // enviar os primeiros bits da corrente
175                     if(I2caRegs.I2CCNT == 0)
176                     {
```

```
178         I2caRegs.I2CCNT = 1;
179     }
180     // Envia a segunda parte da corrente (casas
181     // decimais)
182     else if(I2caRegs.I2CCNT == 1)
183     {
184         I2caRegs.I2CDXR = 1*I2caRegs.I2CDRR;
185         I2caRegs.I2CCNT = 2;
186     }
187     // Se a primeira parte da tensao ja foi enviada,
188     // entao envia a segunda parte
189     else if(I2caRegs.I2CCNT == 2)
190     {
191         I2caRegs.I2CDXR = 0xF2;
192         I2caRegs.I2CCNT = 3;
193     }
194     // Se a corrente ja foi enviada,
195     // entao envia a primeira parte da tensao
196     else
197     {
198         I2caRegs.I2CDXR = 0x03;
199         I2caRegs.I2CCNT = 2;
200     }
201     else
202     {
203         __asm( ".\n\tESTOP0" );
204     }
205 }
206
207
208     if(I2caRegs.I2CISRC.bit.INTCODE == 111)
209     {
210         __asm( ".\n\tESTOP0" );
211     }
212     // Mestre mandou parar
213     if(I2caRegs.I2CISRC.bit.INTCODE == 110)
214     {
215         I2CA_Init();
```

```
216     }
217
218     // Pronto para transmitir
219     if(I2caRegs.I2CISRC.bit.INTCODE == 101)
220     {
221         __asm( ".=ESTOP0" );
222     }
223
224     // Pronto para receber
225     if(I2caRegs.I2CISRC.bit.INTCODE == 100)
226     {
227         __asm( ".=ESTOP0" );
228     }
229
230     // bit Nack
231     if(I2caRegs.I2CISRC.bit.INTCODE == 010)
232     {
233         __asm( ".=ESTOP0" );
234     }
235     PieCtrlRegs.PIEACK.all = PIEACK_GROUP8;
236 }
237 // Fim do arquivo
238 //
```


APÊNDICE B – ARQUIVO EM PYTHON DO RASPBERRY PI

```
1 import tkinter as tk
2 import tkinter.ttk as ttk
3 import tkinter.messagebox as msg
4 import tkinter.filedialog as fdialog
5 import matplotlib.pyplot as plt
6 import pandas as pd
7 import numpy as np
8 import time
9 import smbus
10 from random import gauss
11 from matplotlib import animation
12 from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
13
14
15 class Mainframe(ttk.Frame):
16     def __init__(self, master, *args, **kwargs):
17         # *args junta os argumentos posicionais na tupla args
18         # **kwargs junta os argumentos de palavras chave no
19         # dicionario kwargs
20         super(Mainframe, self).__init__(master, *args, **kwargs)
21
22 class App(tk.Tk):
23     def __init__(self):
24         super(App, self).__init__()
25
26         # Titulo da Janela, dimensao e icone do aplicativo
27         self.title('ConFoSol')
28         self.geometry("")
29         #self.iconbitmap(r'solda.ico')
30
31         # Criar a barra de ferramentas
32         mainMenu = tk.Menu(self)
33         self.config(menu=mainMenu)
34
35         # Cria um menu de Arquivos na barra de ferramentas
36         fileMenu = tk.Menu(mainMenu, tearoff = 0)
37         mainMenu.add_cascade(label='Arquivos', menu=fileMenu)
```

```
37     fileMenu.add_command(label='Salvar', command = self.
38                           saveFile)
39     fileMenu.add_command(label='Salvar_como...', command =
40                           self.saveAsFile)
41     fileMenu.add_command(label='Abrir', command = self.
42                           browseFile)
43
44     # Menu de ajuda
45     helpMenu = tk.Menu(mainMenu, tearoff = 0)
46     mainMenu.add_cascade(label = 'Ajuda',menu = helpMenu)
47     helpMenu.add_command(label = 'Sobre',command = self.
48                           showAbout)
49
50     # Widgets
51     self.connectBtn = ttk.Button(self,text = 'Conectar',
52                                  width = 20, command=self.connect)
53     self.analyzeBtn = ttk.Button(self,text = 'Analisar',
54                                  width = 20, command=self.analyzeGraph)
55     self.graphBtn = ttk.Button(self,text = 'Transmitir',
56                                width = 20, command = self.graph)
57     self.resetBtn = ttk.Button(self,text = 'Resetar',width
58                                = 20, command=self.resetSetting)
59     self.quitBtn = ttk.Button(self,text = 'Fechar',width =
60                               20,command = self.quit)
61
62     self.connectBtn.grid(row=1, column=1, ipady = 5)
63     self.analyzeBtn.grid(row=2, column=1, ipady = 5)
64     self.resetBtn.grid(row=3, column=1, ipady = 5)
65     self.graphBtn.grid(row=4, column=1, ipady = 5)
66     self.quitBtn.grid(row=6, column=1, ipady = 5)
67
68
69
70     # Labels em branco para formatar os espacos entre
71     # widgets
72     ttk.Label(self, text = "").grid(row=0, column=0, ipadx
73                                    =10, ipady = 10)
74     ttk.Label(self, text = "").grid(row=0, column=2, ipadx
75                                    =20, ipady = 10)
76     ttk.Label(self, text = "").grid(row=0, column=5, ipadx
```

```
        =10, ipady = 10)
64    ttk.Label(self, text = "").grid(row=0, column=7, ipadx
        =10, ipady = 10)
65
66    ttk.Label(self, text = "").grid(row=5, column=0, ipadx
        =10, ipady = 10)
67    ttk.Label(self, text = "").grid(row=7, column=0, ipadx
        =10, ipady = 10)
68    ttk.Label(self, text = "").grid(row=8, column=0, ipadx
        =10, ipady = 10)
69    ttk.Label(self, text = "").grid(row=9, column=0, ipadx
        =10, ipady = 10)
70    ttk.Label(self, text = "").grid(row=12, column=0, ipadx
        =10, ipady = 10)
71
72    # Corrente maxima
73    ttk.Label(self, text = "Corrente_maxima_(A)").grid(row
        =1, column=3)
74    self.maxCurrentEntry = ttk.Entry(self, width = 10)
75    self.maxCurrentEntry.grid(row=1, column=4)
76
77    # Término do aumento da transmissão/recepção de
        corrente em segundos
78    ttk.Label(self, text = "Inicio_de_estabilidade_(s)").grid
        (row=2, column=3)
79    self.increaseTimeEntry = ttk.Entry(self, width = 10)
80    self.increaseTimeEntry.grid(row=2, column=4)
81
82    # Início da diminuição transmissão/recepção de corrente
        em segundos
83    ttk.Label(self, text = "Inicio_do_decremento_(s)").grid
        (row=3, column=3)
84    self.decreaseTimeEntry = ttk.Entry(self, width = 10)
85    self.decreaseTimeEntry.grid(row=3, column=4)
86
87    # Corrente estável
88    ttk.Label(self, text = "Segunda_corrente_(A)").grid(row
        =4, column=3)
89    self.secondCurrentEntry = ttk.Entry(self, width = 10)
```

```
90         self.secondCurrentEntry.grid(row=4, column=4)
91
92     # Inicio da estabilidade
93     ttk.Label(self, text = "Segunda_estabilidade_(s)").grid(
94         row=5, column=3)
95     self.stableTimeEntry = ttk.Entry(self, width = 10)
96     self.stableTimeEntry.grid(row=5, column=4)
97
98     # Inicio da segunda diminuicao de corrente
99     ttk.Label(self, text = "Ultimo_decremento_(s)").grid(
100        row=6, column=3)
101    self.lastDecreaseTimeEntry = ttk.Entry(self, width =
102        10)
103    self.lastDecreaseTimeEntry.grid(row=6, column=4)
104
105    # T rmino da transmissao/recepcao em milisegundos
106    ttk.Label(self, text = "Tempo_maximo_(s)").grid(row=7,
107        column=3)
108    self.endTimeEntry = ttk.Entry(self, width = 10)
109    self.endTimeEntry.grid(row=7, column=4)
110
111    # Amostras por segundo
112    ttk.Label(self, text = "Amostras_por_segundo").grid(row
113        =8, column=3)
114    self.samplesPerSecEntry = ttk.Entry(self, width = 10)
115    self.samplesPerSecEntry.insert(0, "5")
116    self.samplesPerSecEntry.grid(row=8, column=4)
117
118    #Botao para atualizar as listas de tempo, corrente e
119    # tensao
120    self.writeBtn = ttk.Button(self, text = 'Alterar', width
121        = 25, command=self.initialParameters)
122    self.writeBtn.grid(row=10, column=4)
123
124    self.bus = smbus.SMBus(1)
125
126    # Cria a janela Mainframe
127    Mainframe(self)
```

```
122     def browseFile(self):
123         self.filePathOpen = fdialog.askopenfilename()
124         data = pd.read_csv(self.filePathOpen)
125         try:
126             self.xValue = data['xValue']
127             self.currentTx = data['currentTx']
128             self.currentRx = data['currentRx']
129             self.voltageRx = data['voltageRx']
130             self.potTx = data['powerTx']
131             self.potRx = data['powerRx']
132         except:
133             msg.showerror('Erro_de_leitura', 'Algum_dos_
134                                     parmetros_nao_foi_encontrado')
135
136     def updateValues(self):
137         try:
138             self.maxCurrent = float(self.maxCurrentEntry.get())
139             self.secondCurrent = float(self.secondCurrentEntry.
140                                         get())
141             self.increaseTime = float(self.increaseTimeEntry.
142                                         get())
143             self.decreaseTime = float(self.decreaseTimeEntry.
144                                         get())
145             self.stableTime = float(self.stableTimeEntry.get())
146             self.lastDecreaseTime = float(self.
147                                         lastDecreaseTimeEntry.get())
148             self.endTime = float(self.endTimeEntry.get())
149             self.samplesPerSec = int(self.samplesPerSecEntry.
150                                         get())
151             if ((self.endTime < self.lastDecreaseTime or self.
152                 lastDecreaseTime < self.stableTime or self.
153                 stableTime < self.decreaseTime or self.
154                 decreaseTime < self.increaseTime)):
155                 msg.showerror('Parmetros', 'O_tempo_maximo_
156                               deve_ser_maior_doque_o_inicio do decreimento
157                               _e_t rmino do incremento!')
158             else:
159                 return
```

```
150     except:
151         msg.showerror('Par metros', 'Algum_dos_par metros
152                         _nao_foi_alterado_corretamente_(separador_de_
153                         decimal_deve_ser_um_ponto_".")')
154     return
155
156     def initialParameters(self):
157         self.updateValues()
158         self.stepSize = 1/self.samplesPerSec
159         self.xValue = [0]
160         self.currentTx = []
161         self.currentRx = []
162         self.voltageRx = []
163         self.j=0
164
165         # Salva os valores de tempo na lista xValue
166         while self.xValue[-1] < self.endTime:
167             self.xValue.append(round((len(self.xValue)*self
168                         .stepSize),4))
169             if self.xValue[-1] > self.endTime:
170                 self.xValue[-1:] = self.endTime
171
172             # Valores de corrente na lista currentTx
173             # Incremento de corrente
174             while self.xValue[self.j] < (self.increaseTime):
175                 self.currentTx.append(round(self.xValue[self.j]*
176                             self.maxCurrent/self.increaseTime, 2))
177                 self.j = self.j +1
178
179             # Corrente maxima constante
180             while self.xValue[self.j] < self.decreaseTime:
181                 self.currentTx.append(round(self.maxCurrent, 2))
182                 self.j = self.j +1
183
184             # Primeiro decremento de corrente
185             while self.xValue[self.j] < self.stableTime:
186                 current = (self.xValue[self.j]*(self.maxCurrent -
187                     self.secondCurrent) + (self.secondCurrent*self.
188                     decreaseTime - self.maxCurrent*self.stableTime))
```

```
        /(self.decreaseTime - self.stableTime)
183    self.currentTx.append(round(current))
184    self.j = self.j +1
185
186    # Segunda corrente estavel
187    while self.xValue[self.j] < self.lastDecreaseTime:
188        self.currentTx.append(round(self.secondCurrent, 2))
189        self.j = self.j +1
190
191    # Ultimo decremento de corrente
192    while self.xValue[self.j] < self.endTime:
193        self.currentTx.append(round(self.secondCurrent*(
194            self.xValue[self.j]-self.endTime) / (self.
195            lastDecreaseTime-self.endTime), 2))
196        self.j = self.j +1
197
198    # Garante que a ultima amostra igual a zero
199    if len(self.xValue) > len(self.currentTx):
200        self.currentTx.append(int(0))
201
202    fig, ax = plt.subplots()
203    ax.cla()
204    ax.plot(self.xValue, self.currentTx, label='Transmissao
205        _da_corrente')
206    ax.legend()
207    ax.set_title('Controle_da_corrente')
208    ax.set_xlabel('Tempo(s)')
209    ax.set_ylabel('Corrente(A)')
210    fig.tight_layout()
211    plt.grid()
212    plt.show()
213
214    def graph(self):
215        plt.style.use('tableau-colorblind10')
216        self.updateValues()
217        figs, (ax1, ax2) = plt.subplots(nrows=2, ncols=1,
218            sharex=True)
```

```
217     self.x = []
218     self.y1 = []
219     self.index=0
220     self.startTime = time.perf_counter()
221     def animate(i):
222         self.i2cTransfer()
223
224         ax1.cla()
225         ax2.cla()
226         ax1.set_title('Comunicacao_I2C')
227         ax1.set_ylabel('Corrente(A)')
228         ax2.set_ylabel('Tensao(V)')
229         ax2.set_xlabel('Tempo(s)')
230         ax1.plot(self.x, self.y1, label='Corrente_'
231                 'transmitida')
231         ax1.plot(self.x, self.currentRx, label='Corrente_'
232                 'medida')
232         ax2.plot(self.x, self.voltageRx, label='Tensao_'
233                 'medida')
233         ax1.legend()
234         ax2.legend()
235         ax1.grid()
236         ax2.grid()
237         figs.tight_layout()
238
239         if self.index == len(self.currentTx):
240             ani.event_source.stop()
241             ax1.fill_between(self.xValue, self.currentRx,
242                             self.currentTx, facecolor = 'green',
243                             interpolate = True, alpha = 0.25)
242             self.powerW()
243             figure, (pax1, pax2) = plt.subplots(nrows=2,
244                                                 ncols=1, sharex=True)
244             pax1.set_title('Potencia_em_Quilowatts')
245             pax1.set_ylabel('Potencia(kW)')
246             pax2.set_ylabel('Potencia(kW)')
247             pax2.set_xlabel('Tempo(s)')
248             pax1.plot(self.xValue, self.potTx, label='
249                         Potencia_enviada')
```

```

249         pax1.plot(self.xValue, self.potRx, label='
250             Pot ncia_medida')
251         pax1.fill_between(self.xValue, self.potTx, self
252             .potRx, facecolor = 'green', interpolate =
253             True, alpha = 0.25)
254         pax2.plot(self.xValue, self.potDif, label='
255             Diferenca_da_pot ncia_medida_e_enviada')
256         pax2.fill_between(self.xValue, 0, self.potDif,
257             where = np.array(self.potDif)>=0, facecolor
258             = 'green', interpolate = True, alpha = 0.25)
259         pax2.fill_between(self.xValue, self.potDif, 0,
260             where = np.array(self.potDif)<=0, facecolor
261             = 'red', interpolate = True, alpha = 0.25)
262         pax1.legend()
263         pax2.legend()
264
265         pax1.grid()
266         pax2.grid()
267         figure.tight_layout()
268         plt.show()
269
270         # Funcao para fazer o envio e recebimento dos dados
271         def i2cTransfer(self):
272             for i in range(self.samplesPerSec):
273                 if self.index == len(self.currentTx):
274                     break
275                 nowTime = time.perf_counter() - self.startTime
276                 while nowTime < self.xValue[self.index]:
277                     nowTime = time.perf_counter() - self.startTime
278                 try:
279                     # Enviar o valor da corrente a ser alterada
280                     MSBTx = (int(self.currentTx[self.index]*100))
281                         >> 8
282                     LSBTx = (int(self.currentTx[self.index]*100)) &
283                         255

```

```
277         self.bus.write_byte(0x10, LSBTx)
278         self.bus.write_byte(0x10, MSBTx)
279     except:
280         print('Problema_na_transferencia')
281     try:
282         # Receber a corrente medida
283         LSBRx = self.bus.read_byte(0x10)
284         MSBRx = self.bus.read_byte(0x10)
285         val = ((MSBRx << 8) + LSBRx)
286         val=val*0.001*gauss(10,1)
287         self.currentRx.append(round(val,2))
288     except:
289         print('Problema_na_recepcao_da_corrente')
290     try:
291         # Receber a tensao medida
292         LSBRx = self.bus.read_byte(0x10)
293         MSBRx = self.bus.read_byte(0x10)
294         val = ((MSBRx << 8) + LSBRx)
295         val=val*0.01
296         self.voltageRx.append(round(val,2))
297     except:
298         print('Problema_na_recepcao_da_tensao')
299         self.x.append(self.xValue[self.index])
300         self.y1.append(self.currentTx[self.index])
301         self.index = self.index + 1
302     return
303
304 def analyzeGraph(self):
305     self.browseFile()
306     figures, (ax01, ax02, ax03) = plt.subplots(nrows=3,
307                                                 ncols=1, sharex=True)
307     ax01.cla()
308     ax02.cla()
309     ax03.cla()
310     ax01.set_title(self.filePathOpen)
311     ax01.set_ylabel('Corrente(A)')
312     ax02.set_ylabel('Tensao(V)')
313     ax03.set_ylabel('Potencia(kW)')
314     ax02.set_xlabel('Tempo(s)')
```

```
315     ax01.plot(self.xValue, self.currentTx, label='Corrente_'
316                 enviada')
317     ax01.plot(self.xValue, self.currentRx, label='Corrente_'
318                 medida')
319     ax02.plot(self.xValue, self.voltageRx, label='Tensao_'
320                 medida')
321     ax03.plot(self.xValue, self.potTx, label='Potencia_'
322                 enviada')
323     ax03.plot(self.xValue, self.potRx, label='Potencia_'
324                 medida')
325     ax01.legend()
326     ax02.legend()
327     ax03.legend()
328     ax01.grid()
329     ax02.grid()
330     ax03.grid()
331
332     figures.tight_layout()
333     plt.show()
334
335
336     def connect(self):
337         try:
338             self.bus.write_quick(0x10)
339             msg.showinfo('Conexao', 'Conectado_com_escravo_no_'
340                         endereco:_0x10.')
341
342         except:
343             msg.showerror('Conexao', 'Escravo_nao_conectado.')
344         return
345
346     def powerW(self):
347         self.potTx = []
348         self.potRx = []
349         self.potDif = []
350         for i in range(len(self.currentRx)):
351             pot = self.currentTx[i]*self.voltageRx[i]
352             self.potTx.append(round(pot/1000,2))
353             pot = self.currentRx[i]*self.voltageRx[i]
```

```
348         self.potRx.append(round(pot/1000,2))
349         self.potDif.append(self.potRx[i] - self.potTx[i])
350
351     def resetSetting(self):
352         self.maxCurrentEntry.delete(0, 'end')
353         self.increaseTimeEntry.delete(0, 'end')
354         self.decreaseTimeEntry.delete(0, 'end')
355         self.stableTimeEntry.delete(0, 'end')
356         self.lastDecreaseTimeEntry.delete(0, 'end')
357         self.secondCurrentEntry.delete(0, 'end')
358         self.endTimeEntry.delete(0, 'end')
359         self.samplesPerSecEntry.delete(0, 'end')
360
361     def saveAsFile(self):
362         self.filePathSave = fdialog.asksaveasfilename(
363             initialdir="/<file_name>", initialfile=
364             ("{}-{}-{}-{})_{}s_({}_{}A_{}SampPerSec".format(
365                 round(self.increaseTime), round(self.decreaseTime),
366                 round(self.stableTime), round(self.lastDecreaseTime),
367                 round(self.endTime), round(self.maxCurrent), round(
368                     self.secondCurrent), self.samplesPerSec),
369                 defaultextension='.csv', filetypes=[("Comma-
370                 Separated_Values", ".csv"), ("All_files", ".*")])
371         pd.DataFrame({'xValue':self.xValue, 'currentTx':self.
372             currentTx, 'currentRx':self.currentRx, 'voltageRx':
373             self.voltageRx, 'powerTx':self.potTx, 'powerRx':self.
374             potRx}).to_csv(self.filePathSave, index=False)
375
376     def saveFile(self):
377         if self.filePathSave != "":
378             pd.DataFrame({'xValue':self.xValue, 'currentTx':self.
379                 .currentTx, 'currentRx':self.currentRx, 'voltageRx':
380                 self.voltageRx, 'powerTx':self.potTx,
381                 'powerRx':self.potRx}).to_csv(self.filePathSave
382                 , index=False)
383
384     else:
385         try:
386             pd.DataFrame({'xValue':self.xValue, 'currentTx':
387                 self.currentTx, 'currentRx':self.currentRx,
```

```
'voltageRx':self.voltageRx, 'powerTx':self.  
potTx, 'powerRx':self.potRx}).to_csv(self.  
filePathOpen, index=False)  
371     except:  
372         msg.showerror('')  
373  
374  
375     def showAbout(self):  
376         # Ajuda sobre o programa  
377         msg.showinfo('Ajuda', 'Para iniciar o programa, aperte  
o botão "Conectar" para estabelecer a conexão I2C  
com o escravo'  
378         'Alterar as correntes e tempos desejados e clicar no  
botão "Alterar" para ver no gráfico as mudanças  
realizadas.'  
379         'Para transferir os dados, conferir se está conectado,  
e clicar no botão Transmitir.'  
380         'Em caso de dívidas, entrar em contato com: Renan.  
Tomisaki@gmail.com')  
381  
382     # Cria e roda o objeto App  
383     App().mainloop()
```