**Universidade de Brasília**
**Faculdade de Tecnologia**

# Integração da instrumentação e controle da aquisição e de armazenamento remoto de dados provenientes de microusinas de geração fotovoltaica *off-grid*

Luiz Felipe Almeida Silva

TRABALHO DE GRADUAÇÃO 2

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Brasília

2024

**Universidade de Brasília**
**Faculdade de Tecnologia**

# Integração da instrumentação e controle da aquisição e de armazenamento remoto de dados provenientes de microusinas de geração fotovoltaica *off-grid*

Luiz Felipe Almeida Silva

Trabalho de Graduação submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação.

Orientador: Prof. Dr. Guilherme Caribé de Carvalho
Coorientador: Prof. Dr. Fernando Cardoso Melo

Brasília
2024

Silva, Luiz Felipe Almeida.
S586i Integração da instrumentação e controle da aquisição e de armazenamento remoto de dados provenientes de microusinas de geração fotovoltaica *off-grid* / Luiz Felipe Almeida Silva; orientador Guilherme Caribé de Carvalho; coorientador Fernando Cardoso Melo. -- Brasília, 2024.
125 p.

Trabalho de Graduação 2 (Engenharia de Controle e Automação) -- Universidade de Brasília, 2024.

1. Automação em *software*. 2. Energia solar. 3. Instrumentação eletrônica. 4. Integração de sistemas. I. de Carvalho, Guilherme Caribé, orient. II. Melo, Fernando Cardoso, coorient. III. Título

**Universidade de Brasília**
**Faculdade de Tecnologia**

# Integração da instrumentação e controle da aquisição e de armazenamento remoto de dados provenientes de microusinas de geração fotovoltaica *off-grid*

Luiz Felipe Almeida Silva

Trabalho de Graduação submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação.

Trabalho aprovado. Brasília, 5 de Julho de 2024:

---

**Prof. Dr. Guilherme Caribé de Carvalho, UnB/FT/ENM**
Orientador

---

**Prof. Dr. Fernando Cardoso Melo, UnB/FT/ENE**
Coorientador

---

**Prof. Dr. Carlos Humberto Llanos Quintero, UnB/FT/ENM**
Examinador externo

---

**Prof. Dr. Mario Benjamim Baptista de Siqueira, UnB/FT/ENM**
Examinador externo

---

**Mestre Darío Gerardo Fantini, UnB/FT/ENM**
Examinador externo

# Agradecimentos

O desenvolvimento desse trabalho representa a consolidação de muito esforço e perseverança durante todo o tempo em que fui aluno do curso de Engenharia Mecatrônica na Universidade de Brasília. Com isso, há algumas pessoas a quem desejo agradecer de antemão.

Em primeiro lugar, a Deus, por ter me acompanhado nessa longa jornada que é o curso de Engenharia Mecatrônica. Não só pela companhia, mas também pelo suprimento de forças e por ter me possibilitado fazer coisas de que nem eu mesmo sabia que era capaz.

Em segundo lugar, aos meus queridos pais, Ildete e Karson, por terem sido uma das "pedras angulares" em minha formação acadêmica, investindo em mim tempo e dinheiro, recursos preciosos na vida de todo ser humano que passa por esta Terra.

Em terceiro lugar, aos professores que tornaram a minha caminhada acadêmica mais leve, com seu excelente trabalho ao ministrar aulas e me motivar na continuação do curso. Quero deixar menção honrosa aos professores Drs. Aida Alves Fadel (que Deus a tenha), Adail de Castro Cavalheiro, Alex da Rosa, Alba Cristina Magalhães Alves de Melo, Andre Von Borries Lopes, Bruno Luiggi Macchiavello Espinoza, Carla Maria Chagas e Cavalcante Koike, Carlos Humberto Llanos Quintero, Deborah de Oliveira, Elaine Cristine de Souza Silva, Guilherme Novaes Ramos, Henrique Cezar Ferreira (obrigado por seu ótimo trabalho na coordenação), Humberto Abdalla Junior, Ivan Marques de Toledo Camargo, João Luiz Azevedo de Carvalho, José Edil Guimarães de Medeiros, Lineu da Costa Araújo Neto, Marcus Vinícius Lamar e Renato Alves Borges.

Quero agradecer ainda mais ao meu orientador Prof. Dr. Guilherme Caribé de Carvalho, por ter sido tão solícito e ter prestado toda a assistência necessária, tanto respondendo às minhas questões como também ao compartilhar conhecimentos e resolver problemas que eu não teria conseguido por conta própria. E claro, pelas aulas ministradas no passado. Agradeço também ao coorientador Prof. Dr. Fernando Melo, por prestar assistência adicional quando requisitei.

Um agradecimento especial ao Prof. Guillermo, da Faculdade do Gama (FGA, Universidade de Brasília), por ter prestado assistência técnica em relação a um problema relacionado ao amplificador diferencial AD620.

Por fim, quero agradecer a todos os demais envolvidos no processo: parentes, amigos, professores que me inspiraram no ensino médio, entre outros. A todos vocês, o meu muito obrigado!

*“Uma jornada de mil milhas começa com um único passo.”*
*(Laozi)*

# Resumo

Em microusinas fotovoltaicas, diversos parâmetros influenciam a geração de energia, como irradiância e temperatura operacional das células fotovoltaicas. Para estudar os efeitos desses parâmetros, são necessárias medições eletrônicas e monitoramento digital. Esses sistemas de geração frequentemente envolvem a interconexão entre diferentes dispositivos e subsistemas. Assim, a necessidade de armazenamento eficiente dessas informações é evidente. Este projeto tem como meta a integração e automação por *software*, desenvolvido em linguagem Python, de um sistema de monitoramento de painéis fotovoltaicos submetidos a diferentes regimes de troca térmica. A instrumentação é interconectada por meio de diversos barramentos de comunicação, como I2C, 1-wire, SPI e RS485, com a implementação do Modbus RTU sobre o padrão RS485. Todo o sistema é gerenciado por um microcomputador *Raspberry Pi*. O resultado obtido é a criação de um sistema autônomo, obtido através de uma camada de gerenciamento e vários processos que realizam tarefas distintas e interagem entre si para gerar arquivos com grandezas relevantes ao estudo dos módulos fotovoltaicos, medidas com uma incerteza tolerável pelo diverso conjunto de sensores do sistema. Esses arquivos são transmitidos periodicamente para um serviço de armazenamento em nuvem.

**Palavras-chave**: Automação em *software*; Energia solar; Instrumentação eletrônica; Integração de sistemas.

# Abstract

In photovoltaic microgrids, several parameters influence energy generation, such as irradiance and the operating temperature of photovoltaic cells. To study the effects of these parameters, electronic measurement and digital monitoring are needed. These generation systems often involve the interconnection of different devices and other subsystems. Therefore, the need for efficient storage of this information is evident. This project aims to integrate and automate, by means of software developed in Python, a monitoring system for photovoltaic panels subjected to different thermal exchange regimes. The instrumentation is interconnected through various communication buses, like I2C, 1-wire, SPI, and RS485, with the implementation of Modbus RTU over the RS485 standard. The entire system is managed by a Raspberry Pi microcomputer. The result obtained is the creation of an autonomous system, obtained through a management layer and various processes that carry out different tasks and interact with each other to generate files with data relevant to the study of photovoltaic modules, which are measured by the system's diverse set of sensors with a tolerable uncertainty. These files are periodically transmitted to a cloud storage service.

**Keywords**: Automation by software; Electronic instrumentation; Solar energy; Systems integration.

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | | |
|---|---|---|
| ADU | *Application Data Unit* | 23 |
| API | *Application Programming Interface* | 39 |
| ARM | *Advanced RISC Machine* | 38 |
| CRC | *Cyclical Redundancy Check* | 48 |
| CS | *Chip Select* | 22 |
| CSV | *Comma-Separated Values* | 50 |
| FAT32 | *File Allocation Table 32* | 36 |
| GPIO | *General Purpose Input-Output* | 46 |
| GUI | *Graphical User Interface* | 63 |
| HDMI | *High Definition Multimedia Interface* | 38 |
| I2C | *Inter-Integrated Circuit* | 21 |
| IP | *Internet Protocol* | 23 |
| LED | *Light-emitting diode* | 39 |
| MISO | *Master In, Slave Out* | 22 |
| MOSI | *Master Out, Slave In* | 22 |
| PDU | *Protocol Data Unit* | 23 |
| RKXXX | Sensor *Rika Sensors*, modelo XXX | 28 |
| RS-485 | *Recommended Standard #485* | 23 |
| RTU | *Remote Terminal Unit* | 35 |
| SCL | *Serial Clock* | 21 |
| SCLK | *Synchronous Clock* | 22 |
| SD | *Secure Digital* | 36 |
| SDA | *Serial Data* | 21 |
| SPI | *Serial Peripheral Interface* | 22 |
| TCP | *Transmission Control Protocol* | 23 |
| USB | *Universal Serial Bus* | 38 |

# Sumário

# 1 Introdução

Nos últimos anos, tem-se discutido amplamente sobre sustentabilidade. Desde a consolidação do uso de combustíveis fósseis, há debates contínuos sobre os impactos dessa matriz no meio ambiente e até quando será possível utilizá-la antes de seu esgotamento. Diante disso, é natural que os engenheiros explorem alternativas que sejam economicamente viáveis a longo prazo e que minimizem os impactos ambientais.

De acordo com Bronzatti e Iarozinski Neto (2008), opções atrativas incluem energia nuclear, energia eólica e energia solar. No entanto, a energia nuclear ainda apresenta limitações ambientais e tecnológicas. Portanto, uma opção atraente para o Brasil são a energia eólica e a solar. A energia solar se destaca como uma opção particularmente interessante, devido à alta disponibilidade de irradiação solar no país (Figura 1), especialmente nas regiões do Nordeste, Goiás e Mato Grosso do Sul, com variação por ano, em média, entre 5500 W/m$^2$ e 6800 W/m$^2$, considerando o período do ano 2005 até o ano 2015 (PEREIRA et al., 2017).

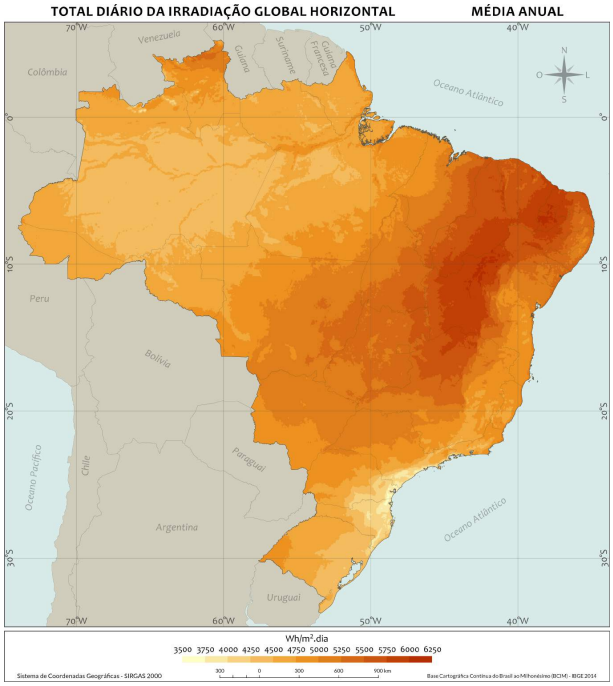



Figura 1 – Mapa da irradiação global no Brasil (média anual).

Fonte: PEREIRA et al. (2017)

Uma forma de converter energia solar em outra forma de energia envolve a concentração de luz solar em um ponto específico, denominada “concentração solar”, método aplicado aos sistemas “heliotérmicos”, que convertem o calor da luz do sol em energia térmica, que por sua vez pode ser convertida em energia elétrica (ou utilizada diretamente). Módulos foto-

voltaicos consistem em outra forma de conversão, convertendo a energia proveniente da luz solar em eletricidade, através do efeito fotovoltaico. São constituídos de células fotovoltaicas, geralmente compostas de silício policristalino ou monocristalino, cuja eficiência é superior a 12% (BARBOSA et al., 2021). No entanto, as perdas neste processo são consideráveis e estão ligadas a diversos fatores, como níveis de irradiação solar, ângulo de inclinação, ângulo de direção relativo ao Sol, temperatura do ambiente de instalação do módulo, condições de resfriamento e outros parâmetros relevantes. Dessa forma, tendo conhecimento de tais fatores, surge um interesse em aprimorar a eficiência dos módulos fotovoltaicos. Esse aprimoramento requer uma compreensão mais profunda de todas as variáveis envolvidas no processo de conversão de energia, como também do impacto das diferentes condições de resfriamento no processo. Por exemplo, a temperatura da célula do módulo fotovoltaico influencia significativamente na eficiência da conversão de radiação solar em energia elétrica. A relevância do trabalho aqui apresentado está em suprir a necessidade de dados para um futuro estudo científico sobre a eficiência dos módulos fotovoltaicos, sob diferentes condições de operação. Neste estudo, será investigada a influência de diferentes variáveis no processo de geração de energia, buscando obter conclusões e instigar a elaboração de estratégias para aprimorar a eficiência.

## 1.1 Objetivos

Considerando o interesse em investigar quais variáveis afetam o processo de geração de energia com a utilização de módulos fotovoltaicos e o que pode ser feito em relação a tais variáveis para o aumento de eficiência dos módulos, é proposto um projeto de pesquisa, com o intuito de analisar o desempenho da geração de energia em microusinas fotovoltaicas sujeitas a diferentes regimes de trocas térmicas. O primeiro regime, em um ambiente terrestre fixo, representa uma situação típica de operação dos módulos fotovoltaicos. O segundo, refere-se a um sistema flutuante sobre corpo d'água, onde água é utilizada como fluido de arrefecimento, aplicada sobre o sistema de diferentes formas. O objetivo é investigar a influência dos fatores externos na eficiência dos módulos e, consequentemente, na geração de energia. De modo a atingir tal objetivo, faz-se necessário a utilização de um sistema que seja capaz de medir as grandezas relevantes envolvidas na conversão de energia de ambos os sistemas. Opta-se por utilizar uma instrumentação eletrônica, cuja saída pode ser obtida e processada por um sistema de monitoramento implementado em um microcomputador, por exemplo.

Naturalmente, certos desafios surgem no que diz respeito à utilização de uma instrumentação eletrônica para medir grandezas relevantes intrínsecas ao ambiente de geração de energia. Isso porque a quantidade de variáveis envolvidas é grande. Dessa forma, se faz necessária a utilização de uma metodologia eficiente para a medição, como quais sensores serão utilizados, de que forma essas informações serão transmitidas e para onde, além de

outras questões. A estratégia principal nesse escopo, como forma adicional às estratégias empregadas em etapas anteriores, é a utilização de um *software* de automação que integre os diferentes sensores presentes no sistema de coleta de dados e a automatize.

Inicialmente, faz-se necessário citar as implementações já existentes, consolidadas em etapas anteriores a esse projeto. No que se refere aos módulos, todos os aspectos mecânicos como o trocador de calor e a estrutura de bombeamento para o sistema flutuante já haviam sido implementados. A instrumentação eletrônica também estava completa (ALMEIDA, 2022), porém apresentava alguns problemas no que se refere ao funcionamento, como sensores sem calibração apropriada e comportamentos indesejados em termos de sinal elétrico, como por exemplo o estágio anterior de um circuito elétrico influenciar em seu próximo estágio, pois a impedância de entrada do estágio sucessor era baixa.

O *software* de automação já se encontrava em fase inicial de desenvolvimento, com alguns módulos de comunicação implementados e um código principal responsável por realizar a comunicação dos sensores em rede Modbus, em fase de *debugging* (busca de solução de problemas como a leitura dos sensores retornando erros (MACIEL, 2021) e taxas de aquisição muito baixas em dispositivos que necessitam uma taxa de aquisição mais alta). Além disso, outros códigos responsáveis por realizar a comunicação com outros barramentos de comunicação estavam prontos e funcionais, mas não integrados e com necessidade de reprojeto. Considerando que o principal objetivo nesse escopo é o desenvolvimento do *software* de automação e a consolidação da estrutura projetada em etapas anteriores desse projeto, destacam-se os seguintes objetivos:

- 1. Consolidação de um *software* de automação para monitoramento de variáveis, integrando todos os subsistemas (isto é, implementações previamente concretizadas, mas não integradas) e automatizando a coleta e envio de dados à nuvem, além de permitir a adição ou remoção independente de variáveis e subsistemas, utilizando o mesmo *software* instalado em diferentes placas *Raspberry Pi*;
- 2. Depuração da instrumentação e das implementações prévias, como também integração de parte da instrumentação eletrônica em uma placa de circuito impresso, de modo a atingir os requisitos funcionais e temporais (isto é, coleta de mais amostras de dados em um dado intervalo de tempo) de toda a instrumentação presente no sistema;
- 3. Implementação de uma camada robusta de gerenciamento para o software de monitoramento, de modo a detectar erros de medição, instrumentação e evitar funcionamentos indesejados no sistema.

## 1.2 Estrutura do texto

A princípio, uma discussão detalhada sobre a tecnologia da instrumentação eletrônica utilizada na estrutura de medição e quais componentes compõem essa instrumentação é feita (Capítulos 2 e 3), para compreender o que se está interessado em integrar e como esses componentes funcionam.

Em seguida discute-se, no Capítulo 4, como a instrumentação dos sistemas em ambiente fixo e flutuante está organizada, quais são as grandezas sendo medidas e como a comunicação entre os diferentes componentes presentes no sistema é feita. O Capítulo 5 apresenta o desenvolvimento detalhado no que diz respeito ao *software* de automação e projetos relacionados à instrumentação eletrônica, além de sua depuração.

Por fim, uma análise sobre os resultados obtidos com o desenvolvimento apresentado no Capítulo 5 e conclusões a respeito do funcionamento do sistema, além de outras considerações relevantes são feitas nos Capítulos 6 a 8.

# 2 Referencial Teórico

O entendimento dos protocolos de comunicação entre dispositivos é algo essencial para compreender como a integração da instrumentação é feita, dado o fato de que os dispositivos presentes no sistema automatizado necessitam de algum meio de transmissão de informação. Nem todos os dispositivos podem ou devem utilizar o mesmo meio de comunicação para transmitir informação, portanto é necessário enunciar alguns critários para a utilização de certos protocolos, além de critérios temporais e parâmetros elétricos. A seguir, são apresentadas características básicas de cada protocolo de comunicação e seu funcionamento. Por fim, um entendimento básico da linguagem de programação *Python* é importante, pois consiste em uma das principais ferramentas utilizadas no sistema computacional (isto é, o computador *Raspberry Pi* com sistema operacional *Linux*) responsável por realizar a gestão dos processos envolvidos na coleta de dados sobre o processo de geração de energia.

## 2.1 Protocolo de comunicação serial

Comunicação serial, também conhecida como interface serial, se refere à implementação de protocolos que enviam dados sequencialmente, *bit* a *bit*, entre dispositivos em um barramento. Assim como qualquer implementação, possui vantagens e desvantagens. Como vantagem, destaca-se que a utilização de um protocolo serial facilita a sincronização da transmissão, algo complexo na comunicação paralela (DAWOUD SHENOUDA DAWOUD, 2020). Outra vantagem é a redução dos custos com cabos, viabilizando comunicações em longas distâncias. Uma desvantagem é a impossibilidade de múltiplos dispositivos utilizarem o mesmo canal de transmissão simultaneamente.

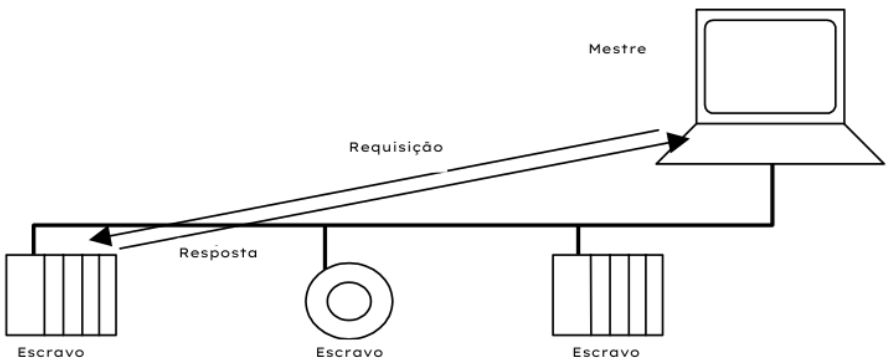



Figura 2 – Comunicação entre mestre e escravo em barramento serial.

Fonte: Adaptado de The Modbus Organization (2006)

Em uma transmissão, o barramento é ocupado por um dispositivo, e outros precisam aguardar até que o dispositivo que está transmitindo finalize a comunicação antes, para que possam então iniciar outra transmissão. Na maioria dos casos, a comunicação serial segue a arquitetura *Master-Slave*, em que um dispositivo age como o "mestre", responsável por enviar comandos aos "escravos", que respondem a esses comandos (Figura 2). Podem-se citar, como exemplos de protocolo serial, os protocolos *1-Wire*, RS485, I2C, SPI e Modbus serial.

## 2.2 Protocolo de comunicação *1-Wire*

O protocolo *1-Wire* (lê-se *one-wire*) é um protocolo de comunicação serial de baixa velocidade *half-duplex* (isto é, o dispositivo ou recebe ou envia, nunca ambos simultaneamente) suportando no máximo 16,3 kb/s (FRENZEL, 2015), dependendo do tamanho do barramento e do número de nós conectados. Desenvolvido pela *Dallas Semiconductor*, tal protocolo apresenta como vantagem a baixa dissipação de potência e a utilização de apenas um fio para energização e transmissão de dados (não obstante, todos os dispositivos precisam de uma conexão com a referência *ground*). Consiste em uma boa escolha de protocolo para aplicações simples, devido ao seu baixo custo e a facilidade de implementação. Com esse protocolo, é possível conectar um máximo de 63 dispositivos como *slave*.

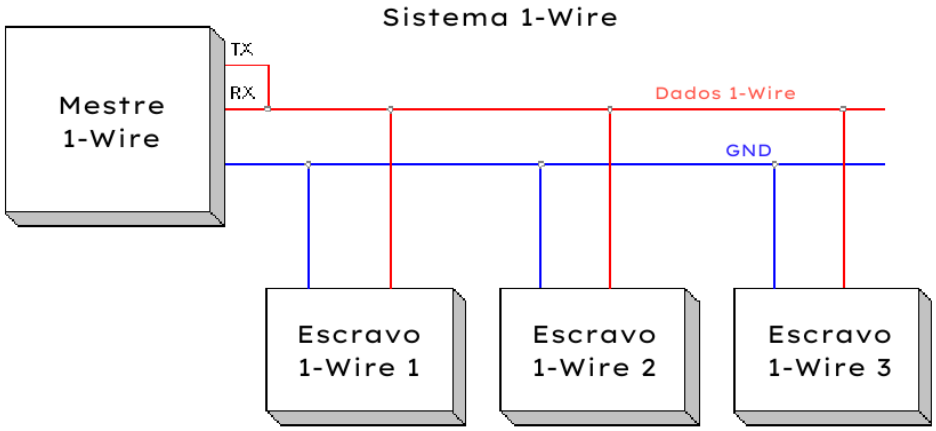



Figura 3 – Exemplo de implementação do protocolo *one-wire*.

Fonte: Adaptado de Cypress Semiconductor Corporation (2014)

Além disso, a maioria dos sensores e atuadores que trabalham nesse protocolo já possuem toda a lógica combinacional implementada internamente. De fato, o mesmo pode ser citado como um exemplo de protocolo *plug and play*.

Para transmitir dados nesse protocolo, os dispositivos são projetados de forma a funcionar com uma técnica de *sample and hold*, isto é, cada dispositivo segura a linha por um período de tempo mínimo (indicando o valor lógico de um *bit*), e após um tempo a

linha é amostrada pelo dispositivo. Se o valor da amostra estiver em nível lógico 1, tal valor corresponde a um *bit* de dados. Em outras palavras, se o mestre deseja receber um *bit* de valor lógico 1, o escravo receberá esse sinal e colocará a linha de transmissão em nível lógico 1. O mestre então recebe o *bit* de valor 1 amostrando a linha em um instante t. TX e RX apenas denotam pinos diferentes, sendo TX o pino de transmissão e RX o pino de recebimento de dados.

## 2.3 Protocolo de comunicação I2C

I2C (*Inter-Integrated Circuit*), refere-se a um protocolo de comunicação serial criado em 1982 pela *Philips Semiconductors*. É um protocolo amplamente utilizado em microcontroladores, principalmente em curta distância, quando o objetivo é realizar a interconexão de circuitos integrados, além da redução de custos para os fabricantes de dispositivos. Permite transmissão em velocidades variando de 100 kb/s (modo padrão) até 3,4 Mb/s (modo alta velocidade) (FRENZEL, 2015).

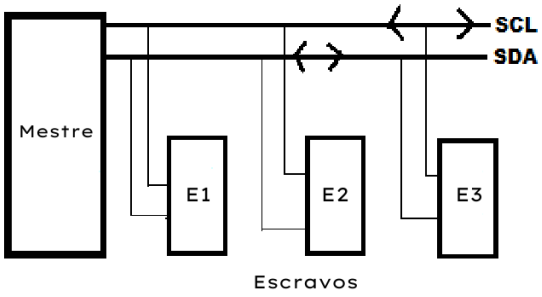



Figura 4 – Esquemático das linhas de transmissão do protocolo I2C.

Fonte: Adaptado de Mankar et al. (2014)

Possui duas linhas de transmissão: SDA (*Serial Data*), responsável pela transmissão de dados entre dispositivos e SCL (*Serial Clock*), responsável por sincronizar a transmissão entre os dispositivos, cujo sinal é geralmente controlado pelo mestre da rede. A comunicação é *half-duplex*. Outra vantagem desse protocolo é a possibilidade de trabalhar com o barramento no modo *multi-master* (isto é, mais de um mestre no barramento), algo desejável em certas aplicações.

O funcionamento é simples: o mestre escolhe um dispositivo através do seu endereço na rede (isso é feito com um sinal de *start*), composto por 8 bits. Em seguida, na detecção do dispositivo, um sinal de *acknowledge* (indicando que o dispositivo de fato existe) é recebido. Os próximos 8 bits se referem aos registradores internos do dispositivo *slave*, que também são finalizados com um sinal de *acknowledge*. Por fim, o dado requisitado é enviado pelo *slave* utilizando os 8 bits restantes. Um último sinal de *acknowledge* é enviado, seguido por um sinal de *stop*.

## 2.4 Protocolo de comunicação SPI

Outro protocolo amplamente conhecido é o protocolo de comunicação SPI (*Serial Peripheral Interface*). Como o próprio nome já sugere, a comunicação também é realizada de forma serial. É um protocolo de alta velocidade, desenvolvido pela Motorola em 1980, com taxas de transmissão variando entre 25 Mbps até 150 Mbps, o que possibilita enviar e receber uma grande quantidade de dados. Diferente dos protocolos anteriormente citados, a transmissão nesse caso é realizada em *full-duplex* (DAWOUD SHENOUDA DAWOUD, 2020), ou seja, é possível enviar e receber dados simultaneamente. Os sinais básicos são quatro:

- CS (*Chip Select*) - Define o dispositivo no qual o mestre deseja estabelecer uma comunicação;
- SCLK (*Synchronous Clock*) - Sincroniza a transmissão entre os dispositivos;
- MISO (*Master In, Slave Out*) - Linha de transmissão dos escravos para o mestre;
- MOSI (*Master Out, Slave In*) - Linha de transmissão do mestre para os escravos.

A Figura 5 demonstra um exemplo de transmissão especificado pelo protocolo:

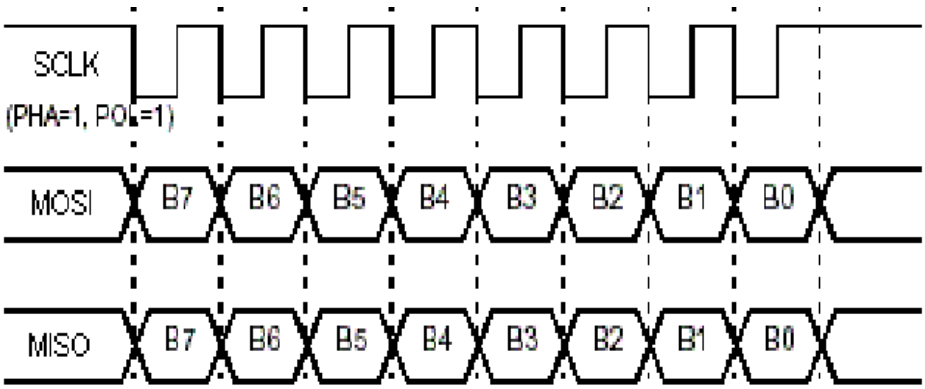



Figura 5 – Esquemático das linhas de transmissão do protocolo SPI.

Fonte: Shingare e Patil (2013)

O funcionamento do protocolo SPI é simples: basta enviar um sinal lógico de ativação na linha de transmissão CS. Após a requisição ao escravo dos dados pelo mestre na linha MOSI, a resposta do escravo no endereço selecionado (isto é, o endereço cujo CS está ativo) é enviada ao mestre pela linha MISO. É possível fazer a requisição e receber o dado de forma simultânea, o que não acontece nos protocolos anteriormente citados.

## 2.5 Interface RS-485

O padrão RS-485 é uma interface de comunicação serial, que define as características elétricas para comunicação serial em dispositivos que utilizam protocolos serial. Seu predecessor é o padrão RS-232. Tal padrão surgiu com a solução dos problemas do RS-232 (curta distância e transmissão em baixa velocidade). Utilizando o padrão RS-485, é possível atingir velocidades de até 10 Mbps a 15,24 m (DAWOUD SHENOUDA DAWOUD, 2020). Além disso, tal padrão apresenta uma boa tolerância a interferências eletromagnéticas, devido à implementação ser em par trançado, o que diminui a ocorrência de erros de transmissão (em cabos com par trançado, os campos magnéticos internos estão em direções opostas, o que causa campo magnético interno líquido igual a zero). É possível conectar até 32 dispositivos. O padrão especifica a transmissão por meio de tensão diferencial, em que uma tensão positiva representa valor lógico 1, e uma tensão negativa, valor lógico 0.

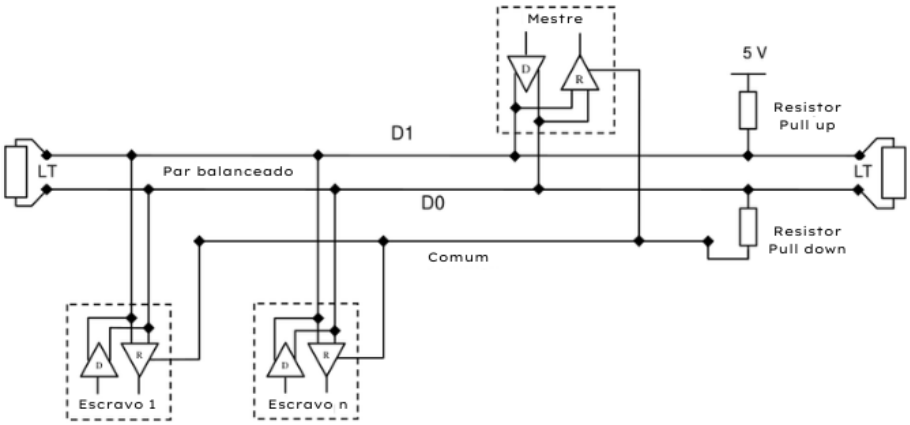



Figura 6 – Exemplo de implementação RS485.

Fonte: Adaptado de The Modbus Organization (2006)

Um ponto importante que pode ser observado na Figura 6 é a presença de resistores de terminação (LT) na linha de transmissão. Sinais lógicos são enviados de um dispositivo a outro por meio de pulsos elétricos. A ausência de resistores de terminação causam a reflexão desses pulsos na chegada do sinal em outro dispositivo, algo bastante indesejado que costuma ser fonte de erros de transmissão. Na presença de tais resistores, tal problema não ocorre.

## 2.6 Protocolo de comunicação Modbus

O protocolo Modbus é um protocolo de comunicação desenvolvido em 1979 pela Modicon, inicialmente para controladores lógico-programáveis, sendo consolidado posteriormente e se tornando um dos principais protocolos utilizados no ramo industrial. Sua

implementação é possível em diversos meios, podendo ser: TCP/IP, RS-232, RS-485, entre outros (THE MODBUS ORGANIZATION, 2012).

Em uma transmissão Modbus, a requisição é feita por meio de um *frame*. A camada elementar desse *frame* (PDU) possui informações básicas de transmissão, sendo essas o código de alguma função, que ocupa 1 byte de espaço no encapsulamento, e um campo de dados, possuindo tamanho máximo de 253 bytes. Campos adicionais podem ser incluídos pela aplicação para o endereço do dispositivo e eventual checagem de erros, o que geralmente acontece, e consiste na camada externa do *frame*, denominada ADU, utilizada pela aplicação.

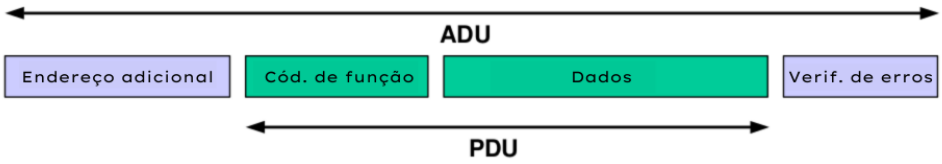



Figura 7 – Estrutura do encapsulamento enviado na rede Modbus.

Fonte: Adaptado de The Modbus Organization (2012)

Após o envio do encapsulamento representado na Figura 7 pelo mestre ao escravo, a resposta é enviada no mesmo formato, porém com o campo de dados tendo a informação requisitada. Em outras palavras, ao requisitar a um sensor de temperatura o valor armazenado em seus registradores, o sensor retornará ao mestre um encapsulamento com o mesmo código da função utilizada e a informação que estava contida nos registradores. Em caso de erros, os campos serão preenchidos com códigos de exceção, como demonstrado nas Figuras 8 e 9.

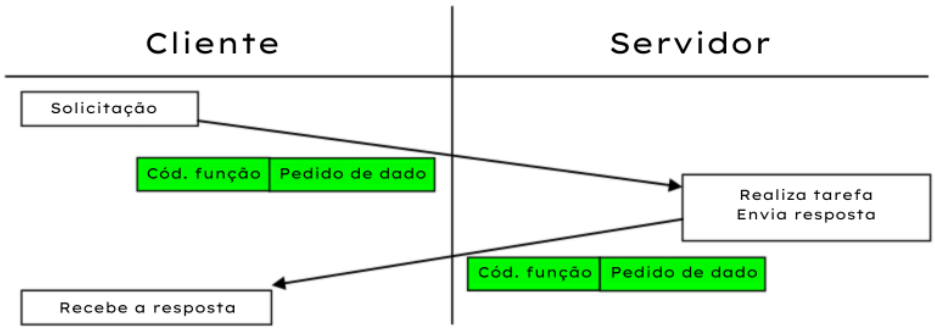



Figura 8 – Exemplo de transmissão.

Fonte: Adaptado de The Modbus Organization (2012)

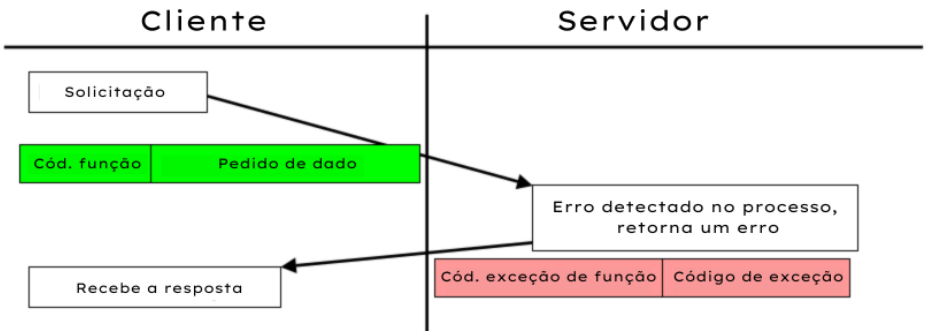



Figura 9 – Exemplo de transmissão com erros.

Fonte: Adaptado de The Modbus Organization (2012)

Os dispositivos que trabalham com o protocolo Modbus seguem um mapeamento lógico de memória, no qual os campos presentes representam diferentes estruturas de dados que podem ser acessadas por códigos de funções específicas, além de apresentar tipos de dados distintos, com tamanhos variados. Cada campo que representa uma estrutura de dados recebe um nome específico. Além disso, o acesso é feito por endereços diferentes. Essas informações representam apenas uma organização lógica da memória do dispositivo, mas não coincidem com as posições de memórias reais presentes no dispositivo, pois a memória não é contígua. A Tabela 1 demonstra como é feito o mapeamento de memória lógica do protocolo Modbus.

| **Tipo de Dado** | **Tamanho** | **Tipo** | **Endereço** |
|---|---|---|---|
| *Coil* | 1 bit | Leitura/Escrita | 1-9999 |
| Reg. Entrada | 16 bits | Leitura | 10001-19999 |
| Reg. Retenção | 16 bits | Leitura/Escrita | 30001-39999 |
| Valor Discreto | 1 bit | Leitura | 40001-49999 |

Tabela 1 – Mapeamento de memória do protocolo Modbus.

Por fim, é importante discutir sobre funções, característica do protocolo Modbus. Suponha que um dispositivo deseja ler a informação contida em outros dispositivos, e essa informação está contida em quatro registradores de retenção. Basta utilizar o código da função de leitura em hexadecimal “0x03”, substituindo o restante dos campos com o endereço do primeiro registrador onde se deseja ler e a quantidade de registradores. Tanto a requisição do primeiro dispositivo quanto a resposta do segundo dispositivo estão demonstrados na Figura 10.

Requisição

| | | |
|---|---|---|
| Código de função | 1 Byte | **0x03** |
| Endereço inicial | 2 Bytes | 0x0000 - 0xFFFF |
| Quantidade de registradores | 2 Bytes | 1 - 125 (0x7D) |

Resposta

| | | |
|---|---|---|
| Código de função | 1 Byte | **0x03** |
| Quantidade de bytes | 1 Byte | 2 x **N*** |
| Valor do registrador | **N*** x 2 Bytes | |

***N** = Quantidade de registradores

Figura 10 – Campos de utilização da instrução *read* no protocolo Modbus.

Fonte: Adaptado de The Modbus Organization (2012)

## 2.7 Linguagem de programação *Python*

*Python* é uma linguagem de programação de alto nível, isto é, em um nível de abstração distante da linguagem binária do processador e mais próxima à linguagem humana, de propósito geral e interpretada (não necessita de um compilador, *software* responsável por transformar um código fonte em um programa executável por algum sistema operacional). A principal vantagem dessa linguagem de programação é a sua legibilidade e facilidade de programação, pois a mesma utiliza mecanismos para evitar códigos de difícil leitura de serem criados e uma simples função ser capaz de executar tarefas que são mais complexas em outras linguagens de programação. Uma desvantagem notória é a dificuldade de se trabalhar em baixo nível (onde é possível manipular palavras binárias), tornando mais complexo ou até mesmo impossível trabalhar com palavras binárias em algumas abordagens. O principal motivo da escolha dessa linguagem são as facilidades que ela oferece, com suas desvantagens sendo toleráveis até certo ponto. A linguagem também apresenta um ponto indesejado que é a sua velocidade de processamento ser reduzida em relação a outras linguagens, o que acontece devido às linhas de programação do código fonte (ou uma versão mais rápida delas) serem traduzidas pelo interpretador em tempo real, o que não acontece nas linguagens compiladas. Felizmente, tal ponto também é tolerável nessa aplicação, mas pode inviabilizar outras.

## 2.8 Considerações finais

A principal motivação desse capítulo é a relevância dos protocolos e das ferramentas para o manuseio da instrumentação eletrônica, demonstrada posteriormente. Sem um entendimento no mínimo rudimentar desses conceitos, torna-se inviável o entendimento pleno de como esses conceitos foram utilizados na metodologia para a implementação da solução do problema de integração proposto na Introdução. Após o que foi apresentado nesse capítulo, já se tem a base necessária para discutir aspectos mais específicos sobre a instrumentação eletrônica utilizada no projeto.

# 3 Instrumentação e implementações

A escolha da instrumentação eletrônica e sensores é feita com base nos objetivos apresentados na Introdução. Os principais fatores que influenciam na conversão de energia são variáveis ambientais, variáveis de atitude (orientação) e variáveis elétricas. Variáveis ambientais, pois fatores intrínsecos ao ambiente em que o módulo está inserido sempre afetam o processo (um módulo fotovoltaico empoeirado transmite menor parcela de irradiação incidente à células voltaicas em seu interior do que um módulo limpo); Variáveis de atitude, devido à orientação dos módulos em relação ao sol reduzir ou aumentar a quantidade de irradiação normal incidente na superfície dos módulos e variáveis elétricas, de modo a detectar qual a relação das outras variáveis com o desempenho da geração de energia. As variáveis elétricas consistem no *feedback* do sistema quando alguma variável previamente mencionada muda. Dessa forma, em cada categoria de variável, é necessário um sensor ou algum instrumento de medição eletrônico. A seguir, são apresentadas as grandezas medidas para cada categoria de variáveis:

- Variáveis ambientais: temperatura do ar, umidade, pressão atmosférica, direção e velocidade do vento, para que seja possível monitorar o clima onde os módulos estão inseridos;
- Variáveis de disponibilidade solar: irradiância no plano inclinado dos módulos, de modo a se obter informações sobre a disponibilidade solar;
- Variáveis de atitude: orientação dos módulos em relação à direção da gravidade e o norte geográfico da Terra;
- Variáveis elétricas: tensão e corrente na associação em série dos módulos fotovoltaicos.

Grande parte dos sensores utilizados são fabricados pela empresa *Rika Sensors*. Instrumentação adicional como módulos de aquisição de termopares para facilitar a obtenção das informações advindas de termopares foi utilizada (MACIEL, 2021), além da implementação do sistema de medição analógico (ALMEIDA, 2022) e dispositivos úteis como um *datalogger* e os controladores de carga, indispensáveis em qualquer sistema de geração fotovoltaico *off-grid*. Por fim, toda essa integração é coordenada por um microcomputador *Raspberry Pi*. Salvo casos em que há menção do tipo de saída de algum sensor ou instrumentação, os sensores apresentados apresentam saídas digitais.

## 3.1 Sensor de concentração de poeira (RK300-02)

O RK300-02 é um sensor de concentração de poeira, que possui dois modelos distintos: *Indoor* e *Outdoor*. Funciona através do princípio do espalhamento a *laser*, onde um feixe emissor de luz *laser* é defletido na presença de uma partícula. A deflexão pode ser medida por um detector específico, cuja saída processada produz informações adicionais. Esse sensor é capaz de detectar partículas de até 1 $\mu$m de diâmetro. Além disso, o sensor tem baixo consumo de energia e rápido tempo de resposta. A Figura 11 mostra o sensor em funcionamento. A Tabela 2 apresenta as características elétricas, além de outras informações, sobre o RK300-02.

Figura 11 – Imagem demonstrando o sensor RK300-02.

Fonte: Produzida pelo autor

| **Parâmetro** | **Informação** |
|---|---|
| Alimentação | 12-24V DC |
| Temperatura de operação | -20 ºC a 50 ºC |
| Alcance | 0-1000 $\mu$g/m$^3$ |

Tabela 2 – Especificações básicas do RK-300-02.

Fonte: Adaptado de Hunan Rika Electronic Technology Co., Ltd (2015d)

## 3.2 Sensor de velocidade e direção de vento (RK120-01)

RK120-01 refere-se a um sensor misto de velocidade e direção do vento, construído com um mecanismo de pás (parte inferior) para a medição de velocidade do vento e uma cauda na parte superior, para medição de sua direção. Seu funcionamento é simples: uma variação na velocidade do vento implicará em maior ou menor velocidade de rotação do eixo de acoplamento do mecanismo das pás, enquanto uma variação na direção altera o ângulo azimutal da cauda superior. Ambas as variações nos mecanismos de pá e cauda geram sinais de saída elétricos em formato analógico, que são convertidos em suas respectivas grandezas

de medição em formato digital e armazenados para posterior processamento. É possível observar mais detalhes sobre esse sensor na Figura 12 e na Tabela 3.

Figura 12 – Imagem demonstrando o sensor RK120-01.

Fonte: Produzida pelo autor

| Parâmetro | Informação |
|---|---|
| Alimentação | 12-24V DC |
| Temperatura de operação | -40 ºC a 70 ºC |
| Faixa de medição | 0-70 m/s (velocidade); 0º-360º (direção) |
| Período de aquisição | 3s |
| Peso | 700g |

Tabela 3 – Especificações básicas do RK-120-01

Fonte: Adaptado de Hunan Rika Electronic Technology Co., Ltd (2015a).

## 3.3 Sensor de temperatura, umidade e pressão (RK330-01)

RK330-01 é um sensor capaz de medir temperatura, umidade e pressão, altamente resistente a ambientes muito úmidos e com altas temperaturas. Além disso, tal sensor tem um tempo de resposta rápido e baixo consumo de energia, como também proteção embutida contra água e raios UV. A Figura 13 e Tabela 4 apresentam informações detalhadas sobre o sensor.

Figura 13 – Imagem demonstrando o sensor RK330-01.

Fonte: Produzida pelo autor

| Parâmetro | Informação |
|---|---|
| Alimentação | 12-24V DC |
| Temperatura de operação | -40 ºC a 80 ºC |
| Faixa de medição | -40 ºC a 60 ºC (Temperatura)<br>0-100% (Umidade)<br>100-1100 hPa (Pressão) |
| Peso | 120 g |

Tabela 4 – Especificações básicas do RK-330-01.

Fonte: Adaptado de Hunan Rika Electronic Technology Co., Ltd (2015e)

## 3.4 Piranômetro (RK200-03)

O RK200-03 é um piranômetro (sensor de irradiância solar), extremamente resistente a ambientes severos. Funciona através do princípio da termopilha, onde junções cobertas por uma camada de alta absorção são colocadas em série. A diferença de temperatura entre junção quente e fria geram uma força eletromotriz proporcional à radiação solar, que pode ser medida pelo sensor. Esse sensor é capaz de medir a irradiância de ondas com comprimento entre 280 nm até 3000 nm. É possível observar mais informações sobre esse sensor na Figura 14 e na Tabela 5.

Figura 14 – Imagem demonstrando o sensor RK200-03.

Fonte: Produzida pelo autor

| Parâmetro | Informação |
|---|---|
| Alimentação | 12-24V DC |
| Temperatura de operação | -40 ºC a 85 ºC |
| Faixa de irradiância | 0-2000 W/m$^2$ |
| Faixa de medição | 280-3000 nm |
| Peso | 580 g |
| Tempo de resposta | Até 13 s |

Tabela 5 – Especificações básicas do RK-200-03.

Fonte: Adaptado de Hunan Rika Electronic Technology Co., Ltd (2015b)

## 3.5 Sensor de irradiância solar (RK200-04)

Diferente do sensor anterior, o RK200-04 é um sensor de irradiância solar que funciona por excitação de elétrons em células de silício. Quando a luz solar atinge uma célula, elétrons são excitados, causando seu movimento. O movimento desses elétrons pode ser detectado por um circuito interno do sensor, responsável por realizar a conversão do movimento de elétrons (corrente elétrica) em grandeza de irradiância solar. Esse sensor, porém, apresenta uma limitação no espectro de detecção de luz, que varia de 300 nm a 1100 nm.

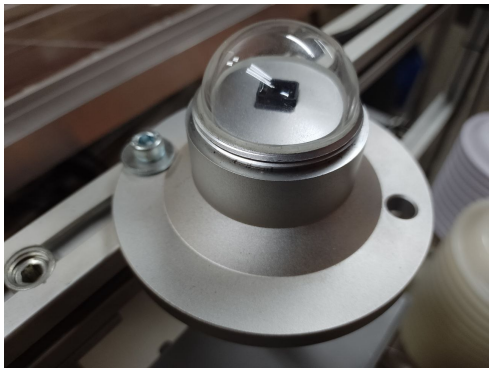


Figura 15 – Imagem demonstrando o sensor RK200-04.

Fonte: Produzida pelo autor

| Parâmetro | Informação |
|---|---|
| Alimentação | 12-24V DC |
| Temperatura de operação | -40 ºC a 80 ºC |
| Faixa de irradiância | 0-1500 W/m$^2$ |
| Faixa de medição | 300-1100 nm |
| Peso | 420 g |
| Tempo de resposta | Até 5 s |

Tabela 6 – Especificações básicas do RK-200-04.

Fonte: Adaptado de Hunan Rika Electronic Technology Co., Ltd (2015c)

## 3.6 Estação meteorológica (RK900-09)

O RK900-09 é uma estação meteorológica de alta precisão com vários sensores integrados, capaz de detectar direção e velocidade do vento, além de temperatura, umidade e pressão. Possui processamento de sinais interno e é capaz de resistir a ambientes severos, além de sua resposta ser consideravelmente rápida. A Figura 16 apresenta o sensor em funcionamento.

Figura 16 – Imagem demonstrando o sensor RK900-09.

Fonte: Produzida pelo autor

| Parâmetro | Informação |
|---|---|
| Alimentação | 24V DC |
| Consumo | 600 mW |
| Temperatura de operação | -40 ºC a 80 ºC |
| Faixa de medição | Vento: 0-40 m/s (velocidade), 0-359º (direção)<br>-40 - 100 ºC (temperatura)<br>0-100% (umidade)<br>1-110 kPa (pressão) |
| Peso | 580 g |
| Tempo de resposta | Até 13 s |

Tabela 7 – Especificações básicas do RK-900-09.

Fonte: Adaptado de Hunan Rika Electronic Technology Co., Ltd (2015f)

## 3.7 Piranômetro LI-200R

O LI-200R é um piranômetro simples, de fácil instalação e capacidade limitada, fabricado pela *LI-COR Enviromental*. A característica mais atraente desse piranômetro é o seu tempo de resposta muito curto (até 1 $\mu$s). Apesar disso, o espectro de detecção desse piranômetro é consideravelmente limitado. O funcionamento do dispositivo é bem simples, a medição é realizada através de um fotodiodo sem filtragem (isto é, o circuito eletrônico não inclui filtros de ruído). O sinal de saída do sensor é analógico, uma corrente da ordem de $\mu$A por 1000 W/m$^2$. Esse sinal é convertido em tensão por um resistor de precisão, que resulta em um sinal da ordem de 10 mV por 1000 W/m$^2$. A Figura 17 demonstra o dispositivo, cujas especificações podem ser observadas na Tabela 8.

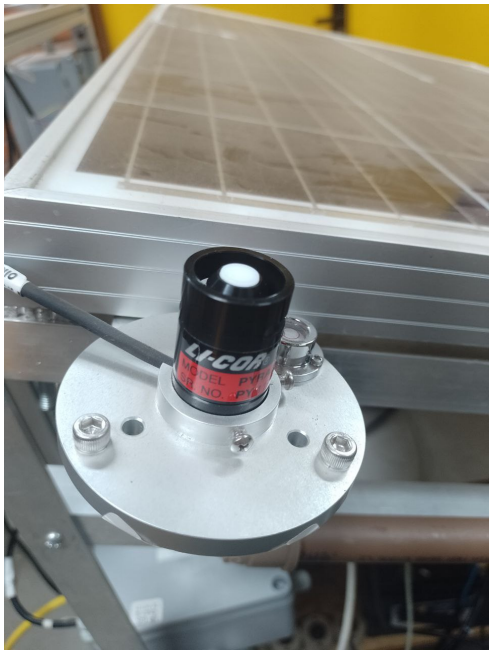



Figura 17 – Imagem demonstrando o piranômetro LI-200R.

Fonte: Produzida pelo autor

| Parâmetro | Informação |
|---|---|
| Faixa de detecção | 0 a 3000 W/m$^2$ |
| Temperatura de operação | -40 ºC a 65 ºC |
| Tempo de resposta | Até 1 $\mu$s |
| Peso | 84g (conjunto) |

Tabela 8 – Especificações básicas do LI-200R.

Fonte: Adaptado de LI-COR Enviromental

## 3.8 Sensor de temperatura DS18B20

DS18B20 é um simples sensor de temperatura, com um sistema de conversão analógico-digital interno, bem útil para aplicações simples de monitoramento de temperatura. Além disso, o sistema de processamento interno do sensor possui vedação, o que possibilita aplicações onde o sensor é imerso em líquidos, que é o que se deseja aqui, já que tal sensor será utilizado para medir as diferentes temperaturas dos fluidos refrigerantes no sistema. A comunicação é realizada em *1-wire*, o que possibilita o sensor ser alimentado e transmitir dados pela mesma linha de transmissão. A Figura 18 é uma imagem ilustrativa do sensor utilizado no sistema.

Figura 18 – Imagem demonstrando o sensor DSB18B20.

Fonte: PIMORONI LTD

## 3.9 Módulo de termopares AM8T

O AM8T é um dispositivo capaz de integrar a aquisição de dados de termopares. Foi idealizado com o intuito de facilitar a aquisição remota de dados de termopares através de interface RS485 e Modbus RTU. É um dispositivo simples, bastando conectar o termopar a um de seus *bornes* para aquisição. Possui resolução menor ou maior, dependendo do tipo do termopar (J, K ou T). É possível observar detalhes sobre esse módulo na Figura 19 e na Tabela 9.

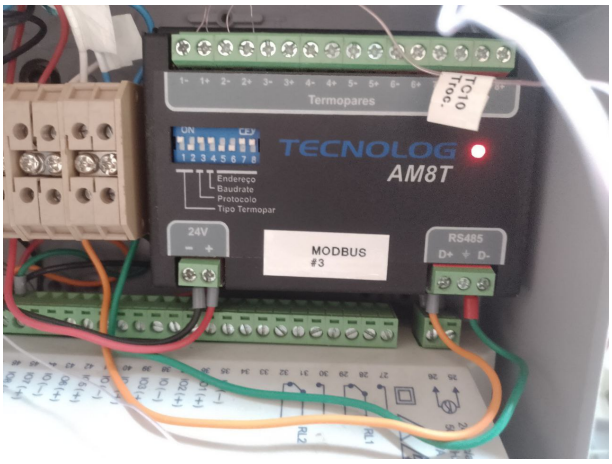



Figura 19 – Imagem demonstrando o módulo de termopares AM8T.

Fonte: Produzida pelo autor.

| Parâmetro | Informação |
|---|---|
| Faixa de medição | 0-790 ºC (Termopar J)<br>0-1100 ºC (Termopar K)<br>0-400 ºC (Termopar T) |
| Alimentação | 24V DC |
| Consumo | 1 W |
| Temperatura de operação | 0 ºC a 60 ºC |
| Período de amostragem | 1 s |

Tabela 9 – Especificações básicas do AM8T.

Fonte: Adaptado de TECNOLOG LTDA (2021)

## 3.10 *Datalogger Fieldlogger*

O *Novus Fieldlogger* é um dispositivo para aquisição e registro de dados. Possui 2 saídas a relé, 8 entradas analógicas, 8 terminais podendo ser de entrada ou saída digitais, além de interface para cartão SD, interface RS485, entrada USB e até serviços que se utilizam do protocolo *Ethernet*. Com esse dispositivo, é possível obter e gerenciar remotamente diversos dados provenientes de diferentes dispositivos conectados ao mesmo. O dispositivo possui uma memória interna de no máximo 2 MB, podendo ser estendida até 32 GB com a utilização de um cartão SD com sistema de arquivos em FAT32. É possível observar o dispositivo na Figura 20.

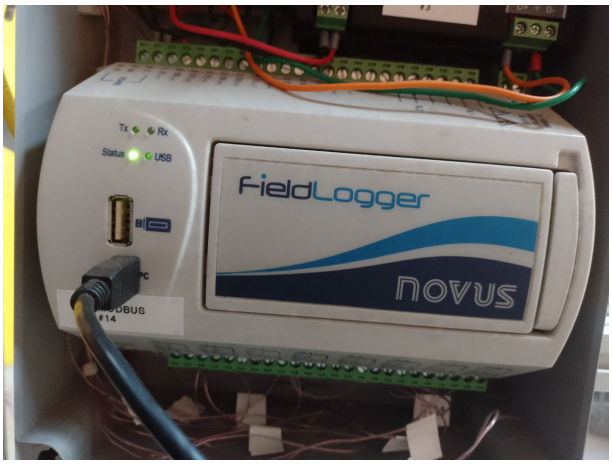



Figura 20 – Imagem demonstrando o *Novus Fieldlogger*.

Fonte: Produzida pelo autor

## 3.11 Advanio W-M1B103

O Advanio W-M1B103 é um módulo de aquisição de termopares, concebido e fabricado pela *Advanio Tech*, capaz de coletar dados provenientes de termopares conectados em

suas entradas analógicas. O dispositivo oferece suporte a comunicação em Modbus RTU e inclui proteção contra surtos de tensão. A Tabela 10 apresenta algumas características básicas desse módulo. É possível ver o módulo em operação na Figura 21.

Figura 21 – Módulo de aquisição de termopares W-M1B103, em operação com termopares acoplados.

Fonte: Produzida pelo autor

| Parâmetro | Informação |
|---|---|
| Resolução | 16 bits |
| Alimentação | 24V DC |
| Taxa de amostragem | 12 amostras/s |
| Consumo | 1,6 W |
| Velocidade máxima | 115,2 kbps |
| Temperatura de operação | -25 °C a 70 °C |
| Peso | 65 g |

Tabela 10 – Especificações básicas do Advanio W-M1B103.

Fonte: Adaptado de Advanio Tech Co., Ltd (s.d.)

## 3.12 Sistema de medição analógico (conjunto SPI)

Um sistema de medição analógico foi previamente implementado para a medição de grandezas relevantes (tensão, corrente) dos módulos fotovoltaicos em trabalhos anteriores desse mesmo projeto (ALMEIDA, 2022). Tal sistema consiste de conversores analógico-digital, amplificadores operacionais diferenciais, sensores de corrente por efeito *Hall* e placas de circuito responsáveis pela calibração do sistema. Nessa implementação, é possivel utilizar o protocolo SPI para coleta de dados após sua obtenção em formato digital pelo estágio final do circuito, a saber, o conversor analógico-digital. A Figura 22 demonstra um diagrama de um circuito elétrico simplificado de como esse circuito foi inicialmente montado.

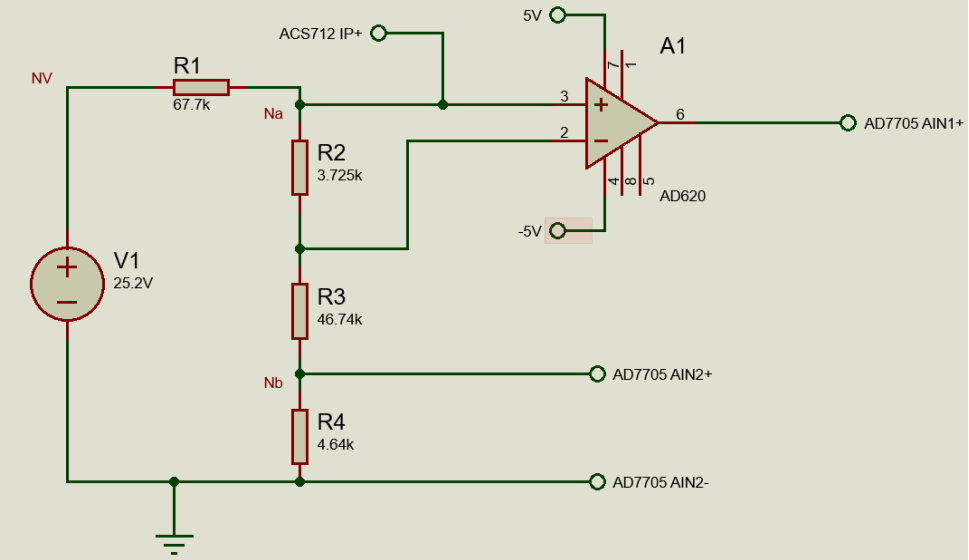



Figura 22 – Imagem demonstrando o esquemático elétrico simplificado do sistema de medição.

Fonte: Produzida pelo autor

Em suma, resistores em configuração “divisor de tensão” são utilizados de modo a reduzir as tensões advindas dos módulos fotovoltaicos associados em série, para que sejam direcionadas ao conversor analógico-digital, cuja saída é posteriormente processada após sua coleta no *Raspberry Pi*. A corrente que passa por essa associação é lida por um sensor de corrente por efeito *Hall*, cuja saída é direcionada a um primeiro conversor AD7705. A tensão do primeiro módulo era previamente direcionada a um amplificador operacional diferencial AD620, com a saída conectada a um segundo conversor AD7705. Isso é feito para que se possa monitorar as tensões de cada um dos módulos separadamente. O segundo divisor de tensão (R3-R4) da Figura 22 é responsável por reduzir a tensão advinda do segundo módulo fotovoltaico, não sendo necessário adicionar outro amplificador operacional para a coleta da tensão de saída. É possível observar detalhes específicos da instrumentação elétrica prévias a esse circuito (como conversores de tensão CC-CC *step-down* e conexões não mencionadas aqui) nos diagramas produzidos pelo orientador desse projeto e por seus ex-participantes no Anexo A. Essa implementação encontrava-se nesse estado no início desse trabalho, porém as mudanças realizadas e projetos referentes a esse circuito serão discutidos posteriormente.

## 3.13 *Raspberry Pi* 4B

O microcomputador *Raspberry Pi* modelo 4B é um vasto sistema computacional implementado em placa de circuito impresso, extremamente portátil e com uma gama de recursos adicionais bem atraente como interface *Wireless*, *Bluetooth*, *Ethernet*, USB, HDMI, entre outros. Além disso, oferece suporte a todos os barramentos utilizados mencionados no Capítulo 2, além de outros barramentos. Inclui sistema operacional baseado em *Linux*

e tem um processador com arquitetura de conjunto de instruções ARMv8, com grande capacidade de processamento (4 núcleos com um *clock* máximo de 1,5 GHz). Em aplicações que requerem portabilidade e capacidade computacional, pode ser citado como uma das melhores escolhas. É possível ver esse componente na Figura 23.

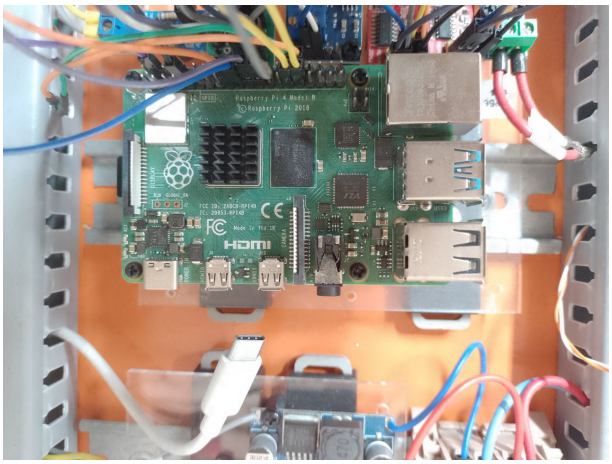


Figura 23 – Imagem demonstrando o microcomputador Raspberry Pi 4B.

Fonte: Produzida pelo autor

## 3.14 Módulo *SenseHat*

O módulo adicional para o microcomputador Raspberry Pi, *SenseHat* é um sistema em placa de circuito impresso que possui um conjunto diversificado de sensores, os quais permitem a medição de pressão, umidade, temperatura, orientação, norte magnético, aceleração, entre outros. Foi inicialmente desenvolvido para utilização na estação espacial internacional (RASPBERRY PI LTD, 2023). É um módulo atraente para muitas aplicações que requerem sensoriamento de posição e grandezas relacionadas a movimento, além de possuir uma boa API (*application programming interface*) em linguagem *Python*. Também inclui um *joystick* para interação e uma matriz LED 8x8 que pode ser utilizada para diferentes propósitos (RASPBERRY PI LTD, 2023). A Figura 24 mostra o componente em funcionamento, onde é possível observar sua matriz de LEDs e os circuitos integrados para cada sensor à direita dessa matriz.

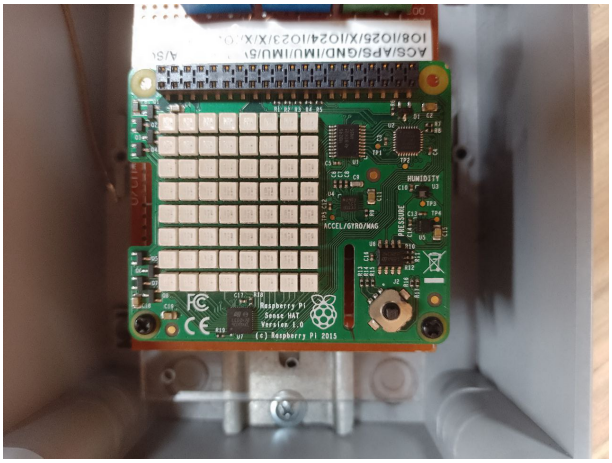


Figura 24 – Imagem demonstrando o módulo *SenseHat*.

Fonte: Produzida pelo autor

## 3.15 Controlador de Carga

Utiliza-se, nesse projeto, um controlador de carga intitulado TRIRON-2210N, fabricado pela *Epsolar Technology*. O controlador de carga é responsável por controlar o fluxo de energia do sistema. Em outras palavras, o dispositivo tem como função controlar o processo de carga e descarga de baterias, tanto aumentando a vida útil das mesmas (a geração de energia durante o dia é variável, pois depende da posição solar relativa e outros parâmetros intrínsecos ao sistema, o que pode prejudicar as baterias), quanto impedindo que a energia contida nas baterias fluam para os módulos solares à noite, quando a tensão dos módulos é baixa. Além disso, ele controla a saída de tensão disponível para determinada carga.

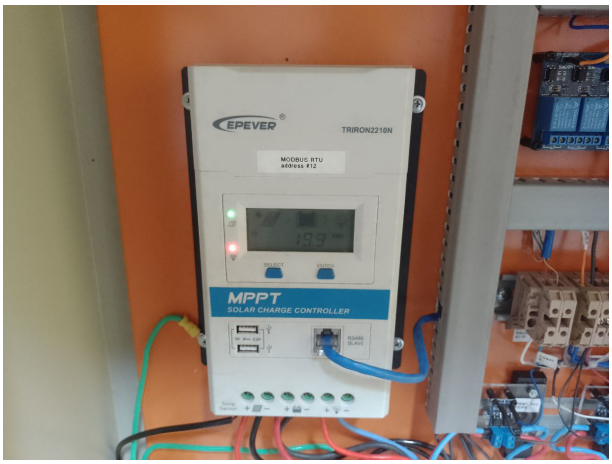



Figura 25 – Imagem demonstrando o controlador de carga TRIRON-2210N.

Fonte: Produzida pelo autor

O dispositivo possui várias características atraentes, o que o torna não só um módulo

básico para gerenciamento do processo de geração de energia, como também um dispositivo adicional para controle de tal processo. Além das funções essenciais de um controlador de carga mencionadas anteriormente, é capaz de fornecer dados úteis em tempo real a respeito da geração. O dispositivo pode ser conectado em rede Modbus para coleta e monitoramento desses dados. Também fornece entrada USB para carregamento de dispositivos.

# 4 Descrição dos Sistemas

Há dois sistemas com os mesmos aspectos construtivos mencionados anteriormente a serem instalados, porém em diferentes ambientes com funcionamento ligeiramente distinto e um conjunto diferente de sensores. O primeiro, em ambiente terrestre, será instalado na laje do Bloco G da Faculdade de Tecnologia da UnB, no campus Darcy Ribeiro. O segundo, em corpo d'água, será instalado na Fazenda Água Limpa, pertencente à Universidade de Brasília.

## 4.1 Sistema Fixo

Tal sistema possui um conjunto reduzido de sensores a serem monitorados. O número de módulos fotovoltaicos também é reduzido, consistindo de apenas dois módulos. O sistema será resfriado apenas por convecção natural. O objetivo é investigar o processo de geração de energia nas condições típicas de instalação de um módulo fotovoltaico. Nesse sistema, monitora-se apenas variáveis triviais do processo de geração de energia, que também serão monitoradas no sistema flutuante.

## 4.2 Sistema Flutuante

Esse sistema possui uma gama mais ampla de sensores ambientais, e consiste no principal objeto de estudo desse trabalho. Nesse sistema, o número de módulos fotovoltaicos é de quatro. A aquisição de dados do conjunto SPI é feita em taxa bem maior (10 Hz) do que no sistema fixo, pois a oscilação das ondas causam mudança de inclinação em relação ao plano horizontal e desvio azimutal em relação ao Sol. O resfriamento nesse módulo se dará de quatro formas diferentes:

- Resfriamento por aspersão de água no *backsheet* do módulo;
- Resfriamento por cortina de água sobre a superfície do módulo;
- Resfriamento indireto por troca de calor com fluido bombeado;
- Resfriamento natural por convecção e radiação (não forçado).

## 4.3 Instrumentação

Considerando o que foi discutido no capítulo anterior, cada sensor coleta um tipo diferente de grandeza física. Essas variáveis precisam ser organizadas para armazenamento e

posterior leitura. A forma mais simples de atingir tal objetivo é se cada sensor tiver uma lista de variáveis lidas associada, com cada variável recebendo um nome específico. As Tabelas 11, 12 e 13 demonstram sensores, barramentos e dispositivos e quais tipos de grandeza os mesmos medem. As tabelas estão organizadas por tipo de protocolo de comunicação utilizado, e os nomes dos parâmetros foram definidos considerando os detalhes da implementação, que serão discutidos posteriormente.

| Sensor | Variáveis lidas | Legenda |
|---|---|---|
| RK300-02 | PM1.0 (ug/m$^3$), PM2.5 (ug/m$^3$), PM10 (ug/m$^3$) | Concentração de poeira por tamanho de partícula, 1 $\mu$m, 2,5 $\mu$m e 10 $\mu$m, respectivamente, em unidades de $\mu$g/m$^3$ |
| RK120-01 | W_speed (m/s), W_dir (graus) | Velocidade e direção do vento |
| RK330-01 | Temp_amb (°C), Umidade_Relativa (%), Pressão (mbar) | Temperatura, umidade e pressão |
| RK200-03 | Irradiância_Termopilha 1 (W/m$^2$) | Irradiância do sensor a termopilha |
| RK200-04 | Irradiância_Célula_silício (W/m$^2$) | Irradiância do sensor de célula de silício |
| RK200-03 | Irradiância_Termopilha 2 (W/m$^2$) | Irradiância do sensor a termopilha |
| RK900-09 | W_speed_2 (m/s), W_dir_2 (graus)<br>Temp_amb_2 (°C), Umidade_Relativa_2 (%), Pressão_2 (mbar) | Velocidade, direção do vento;<br>Umidade, Temperatura, Pressão |
| Advanio_W1 | AD_TC1 - AD_TC8 (°C) | Leitura de termopar por número de conexão |
| FieldLogger | FL_TC1 - FL_TC8 (°C) | Leitura de termopar por número de conexão |
| AM8T | AM_TC1 - AM_TC8 (°C) | Leitura de termopar por número de conexão |
| Controlador12 | PVT1 (V), PVC1 (A), LV1 (V), LC1 (A),<br>Btemp1 (°C), Bsoc1 (%), BV1 (V), BC1 (A) | Controlador 1:<br>Tensão e corrente da associação dos módulos;<br>Tensão e corrente da carga;<br>Estado de carga, tensão e corrente da bateria;<br>Temperatura da bateria |
| Controlador24 | PVT2 (V), PVC2 (A), LV2 (V)<br>LC2 (A), Btemp2 (°C), Bsoc2 (%), BV2 (V), BC2 (A) | Controlador 2:<br>Tensão e corrente da associação dos módulos;<br>Tensão e corrente da carga;<br>Estado de carga, tensão e corrente da bateria;<br>Temperatura da bateria |
| dsb18b20 - 1 | TA_troc_in | Temperatura de entrada de fluido (trocador de calor) |
| dsb18b20 - 2 | TA_troc_out | Temperatura de saída de fluido (trocador de calor) |
| dsb18b20 - 3 | TA_1m | Temperatura da água a 1 metro de profundidade |
| dsb18b20 - 4 | TA_3m | Temperatura da água a 3 metros de profundidade |

Tabela 11 – Variáveis associadas a sensores (Modbus e *1-wire*).

| Sensor | Grandeza |
|---|---|
| Acelerômetro/Giroscópio (LMS9DS1) | *Row, Pitch, Yaw* |
| Magnetômetro (LMS9DS1) | Ângulo em relação à direção norte |
| Ambiental (LPS25H, HTS221) | Pressão, Umidade relativa |

Tabela 12 – Grandezas coletadas com o módulo *SenseHat*.

| Flutuante | Fixo |
|---|---|
| Tensão: módulos 1, 2, 3, 4 | Tensão: módulos 1, 2 |
| Corrente: módulos 1, 2 (Série) | Corrente: módulos 1, 2 (Série) |
| Corrente: módulos 3, 4 (Série) | Irradiância (AD7705) |
| Irradiância (AD7705) | - |

Tabela 13 – Grandezas coletadas utilizando o barramento SPI.

Dessa forma, resta apresentar quais sistemas se utilizam de qual instrumentação para as medições em seus diferentes escopos. A Tabela 14 apresenta uma lista simples, listando quais componentes estão instalados em cada sistema.

| **Componentes instalados por sistema** | |
|---|---|
| **Fixo** | **Flutuante** |
| RK300-02 | RK120-01 |
| RK200-04 | RK330-01 |
| RK200-03 | RK200-03 |
| RK900-09 | Novus Fieldlogger |
| Advanio W-M1B103 | AM8T |
| Controlador 5, 6 | Controlador 1, 2 |
| Barramento SPI | Controlador 3, 4 |
| - | SenseHat |
| - | Barramento SPI |
| - | DSB18B20 (1 a 4) |
| - | LI-200R |

Tabela 14 – Componentes instalados em cada sistema.

## 4.4 Intercomunicação

Tendo como base o que foi apresentado anteriormente, já se têm as informações necessárias para apresentar como a instrumentação é integrada através de seus diferentes protocolos de comunicação. Em outras palavras, é necessário entender como esses diferentes componentes se comunicam entre si, e como essa gestão é feita. A Figura 26 demonstra um esquemático que relaciona os dispositivos e seus respectivos barramentos de conexão com o sistema central em execução no *Raspberry Pi*.

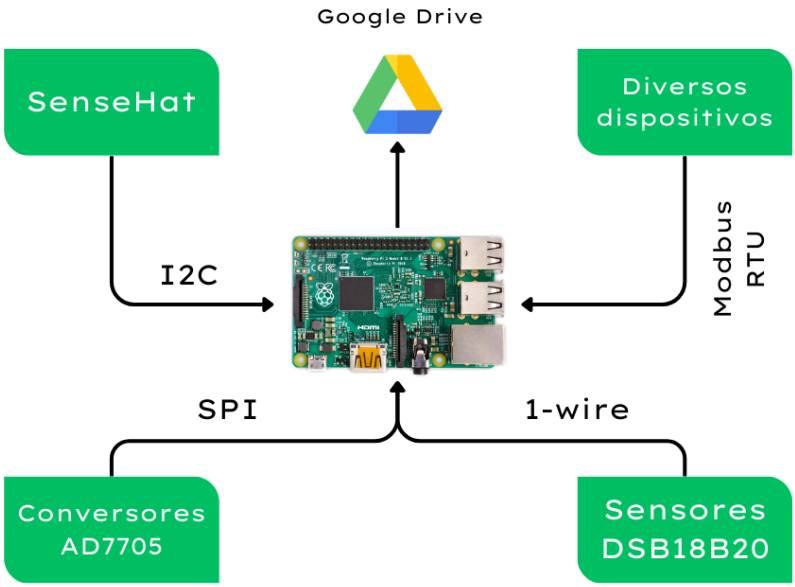



Figura 26 – Esquemático da interconexão entre os diferentes dispositivos por diferentes barramentos.

Fonte: Produzida pelo autor

Durante a coleta de dados, o sistema executa o código contendo as rotinas principais, que serão discutidas com mais detalhes na próxima seção. O esquemático apresentado é para o sistema de módulos em corpo d'água (flutuante), bastando retirar os dispositivos que se comunicam em barramento *1-wire* e I2C para que se obtenha o esquemático do sistema em ambiente terrestre (fixo).

# 5 Metodologia

Com o que foi apresentado nas seções anteriores, é possível discutir com mais detalhes a implementação do software de coleta de dados. A implementação torna-se mais simples com o uso de implementações auxiliares de terceiros para aspectos mais complexos do sistema, o que possibilita que o foco seja direcionado para a implementação das rotinas que resolvam de fato o conjunto de problemas para a solução do problema principal (isto é, a automação do sistema por software), discutido na Introdução. Denomina-se "depuração do sistema" a solução do conjunto de problemas externos ao *software* de automação, intrínsecos à instrumentação utilizada, em busca do funcionamento desejado nesse escopo.

## 5.1 Implementações de terceiros (bibliotecas e *drivers*)

### 5.1.1 Biblioteca para interface SPI: *Spidev*

*Spidev* é uma biblioteca criada para comunicação em baixo nível com dispositivos que se comunicam utilizando o barramento SPI. A biblioteca se utiliza de *drivers* fornecidos por um ambiente em *Linux*. Com as funções implementadas, é possível realizar comunicação direta com um dispositivo conectado no barramento, possibilitando o envio e recebimento de dados em formato binário. A utilização da biblioteca é simples: utilizando a abstração da comunicação para um objeto instanciado em linguagem *Python*, basta definir os parâmetros da comunicação (quais os modos utilizados, quantidade de bits, velocidade) e transmitir os valores binários (podendo ser especificados em hexadecimal) para o dispositivo escolhido no barramento. O Código 8 apresentado no Apêndice B é utilizado de forma mais específica no *driver* de comunicação presente no Anexo B.

### 5.1.2 Bibliotecas RPi.GPIO e pigpio

RPi.GPIO e pigpio são bibliotecas responsáveis pela transferência de dados utilizando os pinos de entrada/saída de uso geral (GPIO) do *Raspberry Pi*. Ambas as bibliotecas oferecem suporte a funções com diferentes propósitos. Porém, a utilização dessas bibliotecas nesse escopo se restringe apenas ao recebimento ou envio de dados, o que pode ser feito mudando o estado lógico desses pinos (ALTO ou BAIXO) de comunicação. A utilização dessas bibliotecas pode ser observada nos códigos do Apêndice B.

### 5.1.3 *Driver* de comunicação para o conversor analógico-digital AD7705

Como anteriormente mencionado no Capítulo 3, um sistema de medição analógico foi implementado utilizando componentes eletrônicos. Um desses componentes, responsável por transformar a saída analógica do sistema em uma saída que possa ser processada pelo microcomputador (ou seja, em formato digital), opera em nível de *hardware*. Dessa forma, há a necessidade de trazer as informações digitais do dispositivo em um formato que possa ser facilmente manipulado por um programa em Python. O código presente no Anexo B (AD770X.py) consiste em uma implementação em Python de um software de comunicação para o conversor analógico-digital, utilizando a biblioteca *Spidev*. Com a utilização desse *software*, é possível abstrair tal componente (isto é, facilitar o seu uso complexo reduzindo a dificuldade para manuseio de poucas funções mais simples) para um objeto simples em linguagem Python, o que facilita o desenvolvimento de *software* que envolva esse dispositivo. O Código 5 apresentado no Apêndice B define os parâmetros elétricos utilizados para teste (que são os pinos de comunicação onde as saídas digitais do conversor analógico-digital estão conectados, além dos ganhos adicionados ao sinal em sua trajetória no circuito elétrico). Em seguida, a leitura é feita com a função de inicialização de ambos os canais de saída do AD7705, e processada imediatamente após a obtenção do valor. A saída do processamento pode ser vista na Figura 27, onde é possível ler os valores de tensão para uma situação onde apenas uma bateria de aproximadamente 24 V está conectada ao circuito elétrico (com o outro ponto de conexão em 0 V), e uma corrente de 300 mA passa pelo caminho que seria a associação dos dois módulos em série.

```
Tensao P1: -3.7463489022452426e-05
Tensao P2: 25.424611394502737
Curr P1_P2: 0.3637369344625019
Tensao P1: -3.7463489022452426e-05
Tensao P2: 25.406777383879675
Curr P1_P2: 0.3400854505226225
Tensao P1: -3.7463489022452426e-05
Tensao P2: 25.421267517510913
Curr P1_P2: 0.3397039749752051
```

Figura 27 – Leitura da tensão e corrente da associação dos módulos.

Fonte: Produzida pelo autor

### 5.1.4 API de controle do módulo *SenseHat*

Uma interface de programação com suporte à linguagem Python foi oficialmente desenvolvida pela *Raspberry Pi Foundation* para o módulo *SenseHat* do *Raspberry Pi*. Com a utilização dessa biblioteca, é possivel trabalhar de forma simples com os recursos que o módulo oferece. A biblioteca apresenta funções que permitem operar com a matriz de LEDs, obter dados dos diferentes sensores e operações com o *joystick*. O Código 6 apresentado no Apêndice B é uma implementação simples de um exemplo de leitura das grandezas utilizadas nesse projeto obtidas pelo *SenseHat* em operação, posicionado com a matriz de LEDs apontada para cima e levemente inclinado em relação ao norte magnético.

```
umidade_relativa(percent): 49.44
pressao(milliBar): 901.70
compass_norte(deg): 315.65
gyro_roll(deg): 359.70
gyro_pitch(deg): 2.15
gyro_yaw(deg): 315.64
accelerometer_roll(deg): 359.69
accelerometer_pitch(deg): 2.21
accelerometer_yaw(deg): 315.64
```

Figura 28 – Leitura das grandezas do módulo *SenseHat*.

Fonte: Produzida pelo autor

### 5.1.5 Módulos de comunicação serial *pySerial* e *MinimalModbus*

O módulo em Python *pySerial* oferece o *backend* (isto é, a complexa implementação em baixo nível, não acessível ao usuário da biblioteca) para comunicação utilizando protocolo serial. Com a utilização dessa biblioteca, é possível facilitar a utilização de um protocolo serial para linguagens de alto nível, através de funções simples que realizam tarefas complexas, facilitando o desenvolvimento de aplicações que se utilizam de comunicação serial. Essa biblioteca é uma dependência (pré-requisito) da biblioteca *MinimalModbus*.

A biblioteca *MinimalModbus* é uma biblioteca desenvolvida com o objetivo de tornar o envio e recebimento de requisições Modbus mais simples. Apesar da comunicação em Modbus por *frames* não ser complexa, as requisições podem ser complicadas de implementar, além de bastante suscetíveis a erros de implementação. A utilização dessa biblioteca restringe a comunicação entre dispositivos a poucas linhas de código, aproveitando a versatilidade de funções específicas, bastando especificar os parâmetros necessários para cada função. Dessa forma, preocupações como cálculo de CRC e tempo de *sleep* até o barramento ser liberado tornam-se irrelevantes. O Código 7 presente no Apêndice B apresenta uma implementação simples de um exemplo de leitura de um dos sensores da rede Modbus e de um parâmetro obtido pelo controlador de carga do sistema, lendo respectivamente as grandezas: Velocidade e direção do vento; Temperatura obtida pelo controlador de carga (multiplicada por 100).

```
MinimalModbus debug mode. Create serial port /dev/ttyUSB0
MinimalModbus debug mode. Will write to instrument (expecting 9 bytes back): '\x06\x03\x00\x00\x00\x02Ä¼' (06 03 00 00 00 02 C5 BC)
MinimalModbus debug mode. Clearing serial buffers for port /dev/ttyUSB0
MinimalModbus debug mode. No sleep required before write. Time since previous read: 1192327718.47 ms, minimum silent period: 2.01 ms.
MinimalModbus debug mode. Response from instrument: '\x06\x03\x04\x00\x00\x00\xadMN' (06 03 04 00 00 00 AD 4D 4E) (9 bytes), roundtrip time
: 2.6 ms. Timeout for reading: 200.0 ms.

Resposta do dispositivo: [0, 173]
```

Figura 29 – Comunicação Modbus: RK120-01.

Fonte: Produzida pelo autor

```
MinimalModbus debug mode. Serial port /dev/ttyUSB0 already exists
MinimalModbus debug mode. Will write to instrument (expecting 7 bytes back): '\x0c\x041\x11\x00\x01n.' (0C 04 31 11 00 01 6E 2E)
MinimalModbus debug mode. Clearing serial buffers for port /dev/ttyUSB0
MinimalModbus debug mode. No sleep required before write. Time since previous read: 6.86 ms, minimum silent period: 1.75 ms.
MinimalModbus debug mode. Response from instrument: '\x0c\x04\x02\x0bUSþ' (0C 04 02 0B 55 53 FE) (7 bytes), roundtrip time: 0.4 ms. Timeout
 for reading: 200.0 ms.

Resposta do dispositivo: [2901]
```

Figura 30 – Comunicação Modbus: Controlador de carga.

Fonte: Produzida pelo autor

Após o recebimento dessas grandezas, basta processar a informação (realizando a conversão da informação para a escala correta, utilizando o fator de conversão encontrado no manual do dispositivo) e armazená-la.

### 5.1.6 Módulo de comunicação para nuvem *PyDrive*

*PyDrive* é uma interface de comunicação que é capaz de automatizar o processo de autenticação e *upload* de dados no serviço de armazenamento em nuvem *Google Drive*®. Com a utilização dessa biblioteca, não é necessária a intervenção do usuário para o envio de dados, algo indispensável nesse escopo, dado que o objetivo é automatizar o sistema de coleta e armazenamento de dados. O Código 9 do Apêndice B demonstra um fragmento do código do software de automação em que a autenticação no *Google Drive* é automaticamente realizada e os arquivos comprimidos e enviados em seguida.

### 5.1.7 Biblioteca padrão da linguagem Python

O termo “biblioteca padrão” de alguma linguagem refere-se às bibliotecas que não necessitam de instalação por parte do programador, uma vez que as mesmas já são reconhecidas pelo interpretador/compilador por padrão. A biblioteca padrão da linguagem Python oferece diversas funcionalidades úteis, como bibliotecas relacionadas a estruturas de dados, bibliotecas matemáticas, compressão de dados, sistema de arquivos, interface gráfica, paralelismo, entre muitas outras. Nesse escopo, algumas bibliotecas padrão foram utilizadas:

- *os*: interface com o sistema operacional.
- *sys*: interface com o interpretador da linguagem
- *time* e *datetime*, para temporização;
- *signal*: tratamento de eventos (sinais enviados a processos);
- *csv*: armazenamento de dados em arquivo *csv*;
- *shutil*: operações de alto nível com arquivos (compactação, por exemplo);
- *json*: operações básicas com arquivos *.json*;

- *tkinter*: interface gráfica;
- *multiprocessing* e *threading*, para implementação de paralelismo e concorrência (será discutido posteriormente).

### 5.1.8 Outras bibliotecas

Algumas bibliotecas foram bem úteis para resolver pequenos problemas. A biblioteca *matplotlib*, por exemplo, é uma poderosa biblioteca de criação de gráficos, que foi utilizada para a implementação do *plotter* do software de automação. Possui como dependência a biblioteca *numpy* (outra biblioteca utilizada), bastante útil que facilita operações matemáticas com vetores e matrizes, por exemplo, além de facilitar operações de mais baixo nível que o Python não suporta nativamente (como inteiro de 16 bits). Por último, a biblioteca Pandas, biblioteca utilizada em análise de dados, foi utilizada no módulo de dados para facilitar a manipulação de arquivos *csv*, no que diz respeito a operações com linhas e colunas, pois o módulo nativo do Python *csv* é muito limitado para tarefas do tipo.

## 5.2 Integração da instrumentação

Como mencionado no Capítulo 1, o software discutido nesse escopo encontrava-se em fase inicial de desenvolvimento, com a parte inicial de comunicação Modbus implementada (mas não funcional). *Scripts* de comunicação com os outros barramentos estavam prontos, mas alguns não satisfaziam os critérios temporais para esse projeto (taxa de aquisição do conjunto SPI de no mínimo 2 Hz). Também era necessário otimizar as implementações já feitas ou em fase de desenvolvimento para legibilidade e execução, além de realizar a integração das mesmas.

A princípio, fez-se necessário reestruturar o código já pronto para que o mesmo fosse mais inteligível e fácil de depurar e adicionar funções. O autor julgou ser uma boa escolha a divisão do *software* por módulos, onde cada módulo possui uma função em específico, com a integração entre os módulos feita por um módulo principal. Dessa forma, cada módulo é responsável por uma parte da instrumentação, sendo o módulo principal encarregado de inicializar e coordenar como tais módulos realizam as diferentes tarefas no sistema. Detalhes sobre as versões finais desses módulos e como a integração é realizada são discutidos posteriormente.

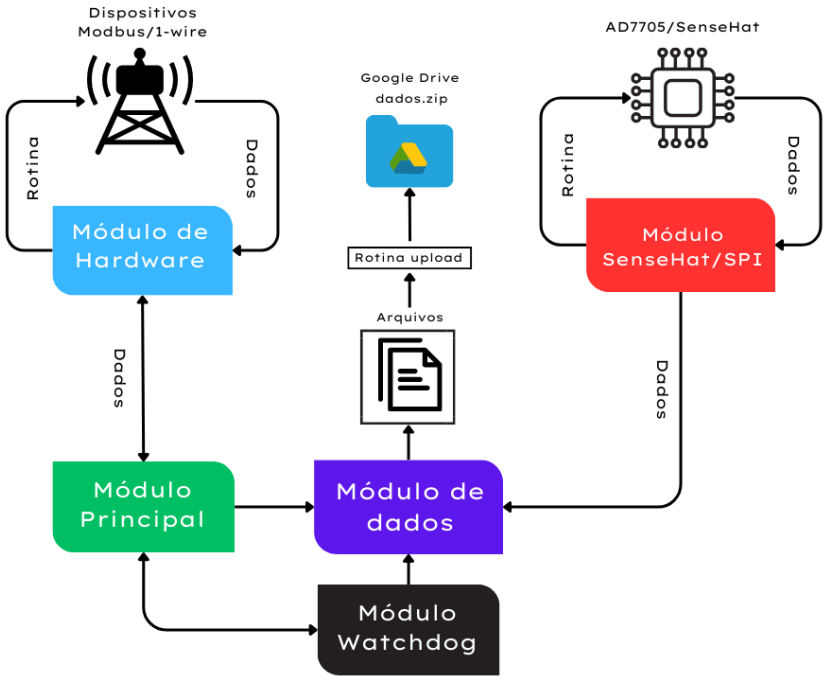



Figura 31 – Estrutura do software e caminho de dados.

Fonte: Produzida pelo autor

A Figura 31 demonstra um esquemático detalhado de como a integração da instrumentação presente nos sistemas é feita. O módulo principal simplesmente coordena o que acontece no sistema, em qual ordem, enquanto que o módulo de *hardware* é responsável pela comunicação *1-wire* e Modbus. A integração dos barramentos SPI e I2C (AD7705 e *SenseHat*) é feita através do módulo *SenseHat*/SPI. Por fim, todo o tratamento de dados como escrita em arquivos e envio dos dados para nuvem é feita pelo módulo de dados. O gatilho que inicia a camada de gerenciamendo do sistema (módulo *Watchdog*) é interno ao módulo principal, que é o primeiro módulo a ser executado, disparando os demais módulos.

## 5.3 Depuração do sistema

### 5.3.1 Rede Modbus

A especificação do protocolo de comunicação serial Modbus estabelece o “tempo ocioso” (*idle time*) entre transmissões como pelo menos o tempo de transmissão de 3,5 caracteres. Em milissegundos, esse tempo de transmissão varia conforme a quantidade de bytes transmitidos e a velocidade de transmissão utilizada no equipamento. Na biblioteca *MinimalModbus*, esse tempo ocioso já é automaticamente calculado (ver Figuras 29 e 30). A Tabela 15 demonstra os tempos corretos para cada taxa de transmissão de um dispositivo em rede Modbus (BERG, 2023). Muitos fabricantes projetam o *driver* de comunicação de

seus dispositivos não levando esse fator em consideração, o que faz com que o tempo ocioso também dependa do dispositivo. Esse era o principal motivo de alguns sensores da rede Modbus não responderem em uma primeira leitura. O problema é resolvido lendo o sensor atual mais vezes em caso de falha (máximo de 5 vezes), o que a contorna e possibilita a leitura normal de todos os sensores da rede.

| ***Bitrate*** | **Tempo de bit** | **Tempo de caractere** | **Tempo de 3,5 caracteres** |
|---|---|---|---|
| 2400 bits/s | 417 us | 4,6 ms | 16 ms |
| 4800 bits/s | 208 us | 2,3 ms | 8,0 ms |
| 9600 bits/s | 104 us | 1,2 ms | 4,0 ms |
| 19200 bits/s | 52 us | 573 us | 2,0 ms |
| 38400 bits/s | 26 us | 286 us | 1,75 ms (1,0 ms) |
| 115200 bit/s | 8,7 us | 95 us | 1,75 ms (0,33 ms) |

Tabela 15 – Tempo ocioso entre transmissões Modbus.

Adaptado de: Minimalmodbus Documentation

### 5.3.2 Taxa de amostragem

Há três conjuntos de dados que devem ser lidos simultaneamente no sistema: O primeiro, em baixa frequência, consiste na leitura dos sensores Modbus, em um período de aproximadamente 20 segundos. No segundo e terceiro conjuntos, é necessária uma leitura em uma taxa de amostragem de no mínimo 2 Hz. Trata-se de um problema com requisitos temporais, o que requer o uso de técnicas e implementações específicas para tal.

Opta-se por utilizar paralelismo ao invés de concorrência para a solução desse problema (paralelismo refere-se à execução de mais de uma tarefa ao mesmo tempo, em núcleos de processamento distintos, enquanto concorrência refere-se a diversas tarefas disputando a utilização de um processador, sendo divididas em *threads*). Dado que o microcomputador *Raspberry Pi* possui quatro núcleos de processamento, a distribuição das tarefas entre os núcleos de processamento de fato é a melhor solução. Detalhes sobre a implementação serão apresentados posteriormente. É importante citar que para que seja possível a obtenção de dados em maior frequência, é necessária a configuração apropriada da comunicação com o conversor AD7705, o que também é discutido posteriormente. É importante mencionar que a decisão de utilizar paralelismo a princípio foi tomada para resolver o problema de coleta de dados em alta taxa, mas isso não implica que concorrência também não é utilizada no *software* de automação. De fato, há processos que possuem diferentes *threads*, tanto para aumento de eficiência, quanto por facilitar a implementação.

### 5.3.3 Módulo SenseHat

Sendo o módulo *SenseHat* um sistema com diversos sensores, os mesmos precisam ser devidamente calibrados para que forneçam medidas corretas. Também é imprescindível que

as informações lidas pelos sensores estejam coerentes com a aplicação. Deseja-se encontrar, para essa aplicação, a direção do norte magnético e a posição do sistema de coordenadas em relação ao centro da placa. A solução para esse problema, além da calibração por *software* do sistema (cujo procedimento pode ser encontrado no site da Raspberry Foundation), é utilizar os próprios sensores contidos no módulo para encontrar a direção dos eixos coordenados e norte magnético. Por exemplo, se um eixo coordenado está com um valor de 1G (1G é aproximadamente 9,81 m/s$^2$), significa que o sentido positivo do eixo em que se mede está apontado naquela direção. O norte magnético pode ser obtido com a medição do valor *compass* (inclinação em graus em relação ao norte magnético), utilizando a API projetada pela *Raspberry Pi Foundation*. Dessa forma, conclui-se que os eixos coordenados se encontram na seguinte posição:

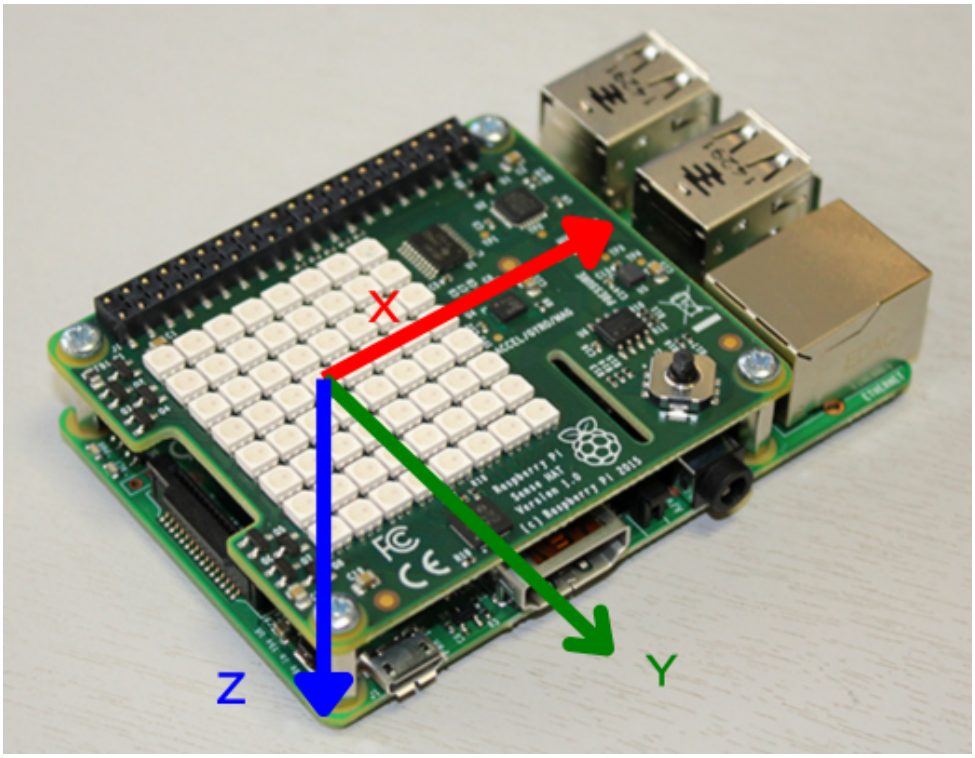



Figura 32 – Eixos coordenados após identificação.

Fonte: Adaptado de: Raspberry Pi Foundation

A direção do norte magnético é obtida usando as medições do magnetômetro presente na placa de circuito. Um eixo coordenado com o eixo X alinhado com o norte magnético apresenta uma medida de defasagem do ângulo azimutal (*yaw*) de 0°. Apesar disso, é importante frisar que o norte geográfico da Terra não está exatamente alinhado com seu norte magnético, requerendo uma pequena correção. De acordo com um modelo magnético implementado pela National Center for Enviromental Information, a inclinação do norte magnético para 2024 em 15,9432° S e 47,9489° W (local de instalação da plataforma) é de 22,13°, que seria o valor a ser subtraído da leitura do norte magnético para a obtenção do norte geográfico

desejado para essa aplicação. A incerteza nesse valor é de 0,38°, e o valor tem um desvio incremental por ano de aproximadamente 0,09°. Isso foi considerado inicialmente na implementação do *software* de automação, mas consiste em uma escolha melhor a correção em análise dos dados coletados.

### 5.3.4 Configuração do *driver* de comunicação AD7705

O conversor AD7705 possui registradores responsáveis pela gerência do funcionamento do circuito integrado. Em outras palavras, a conversão de um sinal analógico para digital pode ser influenciada de acordo com os valores binários presentes nos registradores desse circuito integrado. De fato, o papel do *driver* de comunicação do AD7705 é realizar a manipulação desses registradores através da biblioteca anteriormente discutida *Spidev*, de acordo com as funções utilizadas.

Dentre os registradores do conversor, apenas dois são de relevância nesse escopo: O registrador de *setup* (*setup register*), e o registrador de *clock* (*clock register*). Com uma leitura detalhada do *datasheet* do dispositivo, informações importantes devem ser consideradas para o funcionamento desejado aqui. Deseja-se uma alta taxa de aquisição de dados (a maior possível) e valores que sejam coerentes com o valor analógico que vem do circuito eletrônico anterior ao conversor analógico-digital, portanto, de acordo com o *datasheet* do componente (ANALOG DEVICES, 2006):

- A cada leitura, uma calibração deve ser realizada no canal;
- Se o oscilador do dispositivo possui um *clock* de ou superior a 4,915 MHz, o campo CLK do registrador *clock* deve estar em ALTO; Esses campos definem as correntes internas corretas para que o circuito integrado funcione conforme a especificação.
- Para que o filtro *notch* interno ao circuito integrado tenha frequência de corte em 65,5 Hz (muito distante da taxa de aquisição de aproximadamente 10 Hz) e a taxa de atualização de dados seja de aproximadamente 250 Hz, os campos FS1 e FS0 do registrador de *clock* devem estar em ALTO e BAIXO, respectivamente. A taxa pode ser aumentada ou diminuída com alteração desses campos, mas isso necessariamente altera também a frequência de corte do filtro.

Além disso, o dispositivo também possui dois aspectos importantes a se considerarem: Faixa de valores para entrada e impedância de entrada. Com alimentação de 3,3 V sem *buffer*, o dispositivo permite valores de entrada de 0 a 3,3 V, mas a impedância de entrada não é muito grande e pode haver efeito de carregamento no circuito de entrada. Caso seja utilizado o modo com *buffer*, a impedância de entrada é muito alta, mas o dispositivo só admite entradas de 0 a 1,8 V, para uma alimentação de 3,3 V.

No registrador de *setup*, o pino de FSYNC serve para aplicar um *reset* no filtro interno

ao circuito integrado, impedindo que o dispositivo processe entradas. Os três bits de G2, G1 e G0 servem apenas para escolher um ganho que varia em potências de 2, sendo o mínimo de 1 e o máximo de 128. Como não se deseja um ganho diferente de 1, pode-se deixar todos esses bits em zero. Quanto à calibração, opta-se pelo modo *self-calibration*, que é uma sequência de calibração simples nos dois canais de entrada do conversor.

Dessa forma, os registradores de *clock* e *setup* podem ser configurados dentro do *driver* de comunicação do AD7705 conforme mostra a Tabela 17 e 18. A Tabela 16 descreve qual a função de cada campo nesses registradores.

| Campo | Função |
|---|---|
| MD1, MD0 | Palavra que configura a calibração<br>00: Sem calibração<br>01: *Self-calibration*<br>10: *Zero-scale system calibration*<br>11: *Full-scale system calibration* |
| G2, G1, G0 | Palavra cujo valor em decimal na base 2 corresponde ao ganho<br>aplicado na conversão (1 para valor igual a 0, 128 para valor igual a 7) |
| 'B/U | 0: Modo bipolar<br>1: Modo unipolar |
| BUF | 0: Modo *buffered*<br>1: Modo *unbuffered* |
| FSYNC | Sincronizador de filtro. Se em valor 1, a lógica de controle dos filtros e calibração são ativados.<br>Obs: O dispositivo não funcionará enquanto esse bit não voltar ao estado 0. |
| CLKDIS | Se em valor 1, desabilita a saída vinda da entrada de *clock* externo. |
| CLK | Seletor de frequência de clock. Deve corresponder à frequência do oscilador do dispositivo.<br>0: Frequências de 1 MHz e 2 MHz<br>1: Frequências de 2,457 MHz e 4,915 MHz |
| CLKDIV | Divisor de clock do dispositivo mestre. Divide o clock do dispositivo mestre por 2<br>antes de ser utilizado pelo dispositivo (se em valor lógico 1). |
| FS1, FS0 | Bits seletores de taxa de atualização, valor máximo de 500 Hz<br>Consultar referências para mais detalhes. |

Tabela 16 – Descrição de campos dos registradores de *clock* e *setup*.

Adaptado de Analog Devices (2006)

| MD1 | MD0 | G2 | G1 | G0 | 'B/U | BUF | FSYNC |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

Tabela 17 – Palavra binária do registrador *setup register*.

| ZERO | ZERO | ZERO | CLKDIS | CLKDIV | CLK | FS1 | FS0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

Tabela 18 – Palavra binária do registrador *clock register*.

Dessa forma, o registrador de *setup* seleciona o modo de *self-calibration* com os *bits* MD1 e MD0 em 01, o ganho unitário, o modo sem *buffer*, e operação no modo bipolar, sem aplicar *reset* nos filtros.

Quanto ao registrador de *clock*, os três primeiros *bits* devem ser obrigatoriamente

zero, por especificação do fabricante. O *bit* CLKDIS serve apenas para desabilitar o pino de *clock* acessível ao usuário (irrelevante nesse escopo, pois não se está utilizando o AD7705 como fonte de *clock* para outros dispositivos), e os *bits* restantes são configurados de modo a tornar a taxa de aquisação a mais alta possível de acordo com o *clock* do circuito integrado, o que já foi discutido.

### 5.3.5 Correção do circuito de condicionamento

O circuito de condicionamento dos sinais de tensão vindos dos módulos fotovoltaicos apresentavam valores incongruentes ao chegar nos estágios de amplificador operacional, após medição digital nos conversores AD7705. De acordo com os cálculos realizados utilizando as Equações 1 e 2 e considerando o circuito da Figura 22, era esperado um valor real para $V_1$ de 13,94 V, e para $V_2$ de 9,62 V, após conectar uma fonte de alimentação como entrada de aproximadamente 25,3 V. Porém, os valores observados com medições foram, respectivamente, 21 V e 4 V, o que não está de acordo com os cálculos realizados.

$$R_1 = 67{,}7\,k\Omega \quad R_2 = 3{,}725\,k\Omega \quad R_3 = 46{,}74\,k\Omega \quad R_4 = 4{,}64\,k\Omega$$
$$R_T = R_1 + R_2 + R_3 + R_4 = 122{,}805\,k\Omega$$
$$V = 25{,}3V \quad G_1 = \frac{R_2}{R_1} \quad G_2 = \frac{R_4}{R_3}$$

$$V_1 = \frac{R_2}{R_T G_1} V \tag{1}$$

$$V_2 = \frac{R_4}{R_T G_2} V \tag{2}$$

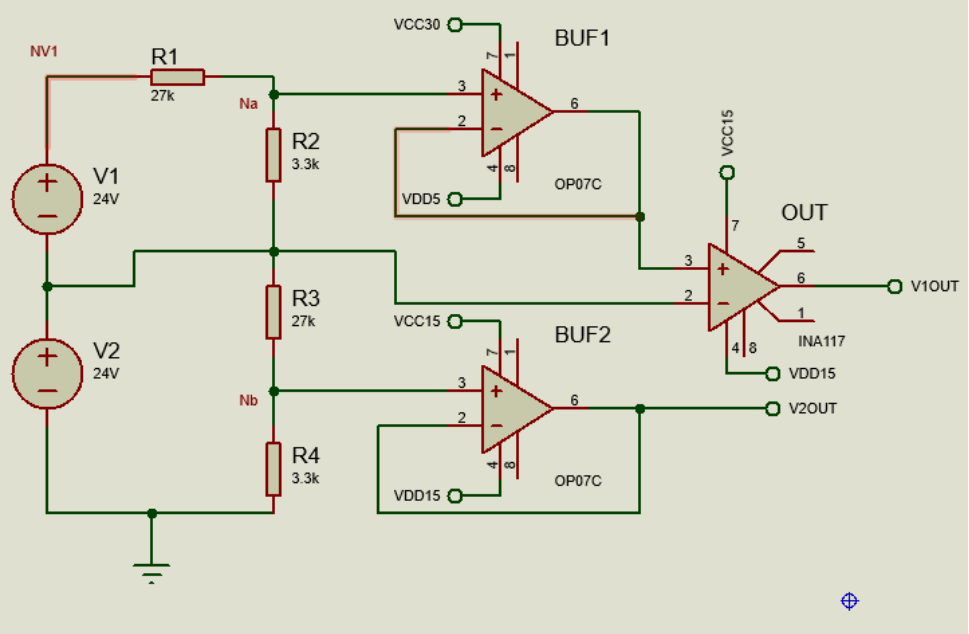



Figura 33 – Esquemático elétrico simplificado do sistema de medição final.

Fonte: Produzida pelo autor

Após exaustivos testes e recebimento de consultorias, constatou-se que o problema acontecia devido à alimentação do amplificador operacional AD620, que não admite, em suas entradas não inversora e inversora, valores de tensão (em módulo) superiores às tensões usadas em sua alimentação (a alimentação era de 5 V, porém a tensão na entrada não-inversora era muito maior que isso). Ao trocar a alimentação para 12 V e garantir que a tensão na entrada não-inversora no amplificador operacional superior fosse inferior a esse limite, os valores voltaram a ser os esperados. Porém, os módulos AD620 não suportam a tensão de 25,43 V no nó Na (Figura 33), causada pela associação em série dos módulos fotovoltaicos, após a queda de tensão do resistor R1. Ou seja, foi necessária a troca de todos os amplificadores operacionais utilizados no circuito e a utilização de amplificadores operacionais que suportassem tensões de modo comum maiores que pelo menos 26 V. Como a esse ponto a instrumentação desse circuito já consistia de várias pequenas placas de circuito e haviam muitos pontos possíveis de falha, foi proposto um projeto de uma placa de circuito que fosse capaz de integrar a maior parte dessa instrumentação em uma única placa. Ademais, um amplificador operacional foi adicionado na parte inferior do circuito (entrada não-inversora conectada no nó Nb, ver Figura 33), pois o conversor CC-CC utilizado no projeto da placa de circuito integradora (que será discutido posteriormente) requer um consumo de no mínimo 10% da carga total conectada à saída do equipamento. Dessa forma, o circuito de condicionamento sem correção de efeitos de ruído e outros fatores como alimentação dos componentes está demonstrado na Figura 33. O amplificador operacional OP07C suporta tensões de entrada com valor positivo de até no máximo 36 V (fonte de alimentação simples positiva), tal valor não deve ser ultrapassado, ou o componente será danificado. Portanto, a

tensão de alimentação para o amplificador operacional conectado no nó Na da Figura 33 foi fixada em 30 V (mas dificilmente atinge esse valor no pior caso). Após algumas simulações em *software*, constatou-se que a alimentação negativa do amplificador operacional do nó Na precisa ter um valor um pouco menor que 0 V. Para fins de conveniência, fixou-se esse valor em -5 V. Esses valores são seguros, desde que a diferença entre eles não ultrapasse 36 V. Os outros amplificadores operacionais admitem tensão diferencial nas entradas menor que 15 V (valor absoluto), mas essas tensões não chegam nem mesmo a atingir metade desse limite. Portanto, fixou-se as tensões de alimentação para o amplificador INA117 e OP07C do nó Nb da Figura 33 em 15 V, com fontes de alimentação simétricas.

## 5.4 Projeto e implementação de uma placa de circuito integradora

Os estágios do circuito de condicionamento dos sinais de tensão vindos dos módulos fotovoltaicos encontravam-se com grandes incongruências no que se refere a suas medidas, como já foi mencionado anteriormente. Os divisores de tensão presentes nas entradas dos módulos fotovoltaicos também foram implementados em pequenas placas de circuito impresso, denominados *placas P25*. De modo a facilitar a integração dessa instrumentação, um projeto de placa de circuito impresso foi proposto, após a solução dos problemas apresentados na Seção 5.3. O circuito de condicionamento projetado e simulado por meio do *software PSpice for TI* pode ser visto na Figura 34.

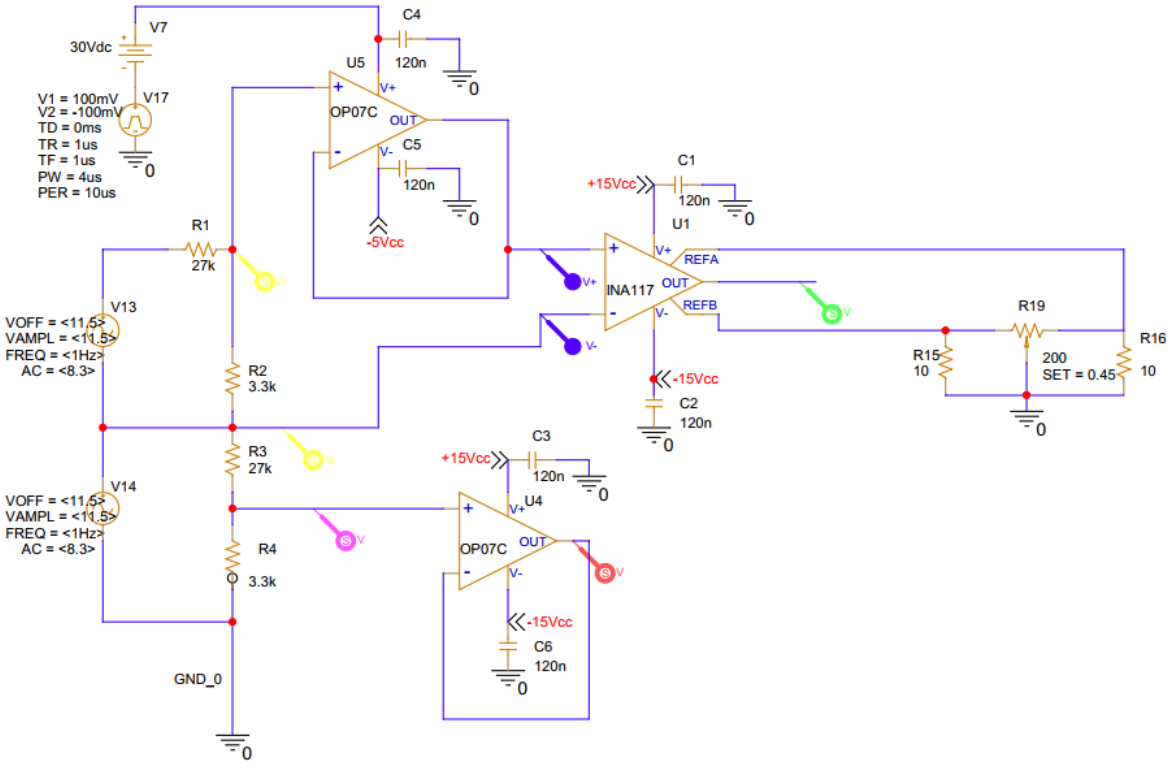



Figura 34 – Esquemático do circuito de condicionamento 1.

Fonte: Guilherme Caribé de Carvalho (orientador do projeto)

Os *buffers* de entrada são responsáveis por cancelar os efeitos de carregamento no próximo estágio do circuito. Utiliza-se um amplificador operacional INA117 no primeiro módulo da associação em série, conectado ao canal 1 do conversor AD7705. No segundo módulo da associação em série, cujo terminal negativo está conectado ao terra comum do sistema, emprega-se um amplificador operacional OP07C, com sua saída conectada ao canal 2 do conversor AD7705. O terminal negativo do primeiro módulo da associação conecta-se ao terminal positivo do segundo módulo, resultando em uma tensão de modo comum próxima à tensão do segundo módulo da associação. A necessidade do uso do amplificador diferencial para medir a tensão do primeiro módulo decorre da exigência do circuito integrado AD7705 de que os terminais de entrada negativos dos dois canais diferenciais estejam conectados ao terminal de referência da fonte que os alimenta, no caso, o terra comum do sistema. Os divisores de tensão nas entradas não mudaram de função, apenas foram integrados à placa. As tensões de alimentação dos amplificadores operacionais +30 V, +15 V, -15 V e -5 V são obtidas a partir de tensões de alimentação externas à placa (+30 Vcc e +5 Vcc). A tensão de 5 V é transmitida para um conversor CC-CC A0515SDL-2W *step-up* de 5 V para +15 V e -15 V, tensões geradas para os amplificadores INA117 e OP07C inferior, como também para um inversor ICL7660SCPAZ de 5V para -5V, tensão gerada para o amplificador OP07C superior. Capacitores de desacoplamento estão presentes por todo o circuito de modo a reduzir ruídos

eletromagnéticos, vindo de várias partes do circuito. Dessa forma, o circuito final para a placa integradora e uma visão 3D projeto final da placa podem ser vistos nas Figuras 35 e 36. O esquemático completo da Figura 35 pode ser encontrado no Apêndice A

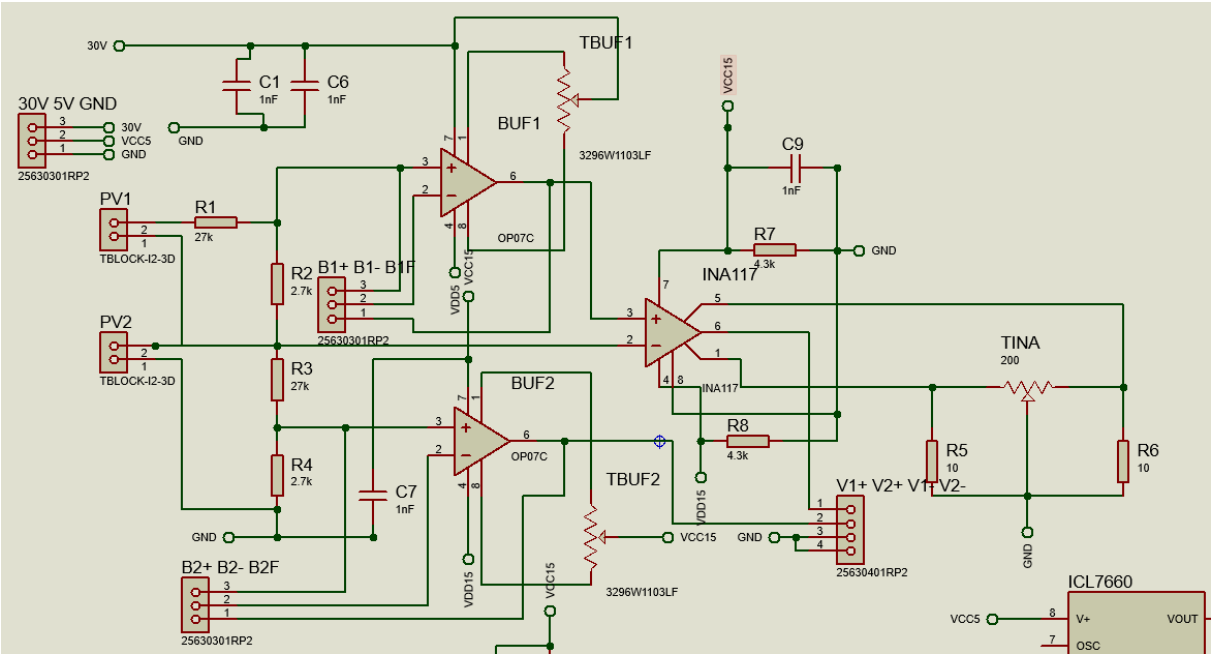



Figura 35 – Esquemático do circuito de condicionamento 2.

Fonte: Produzida pelo autor

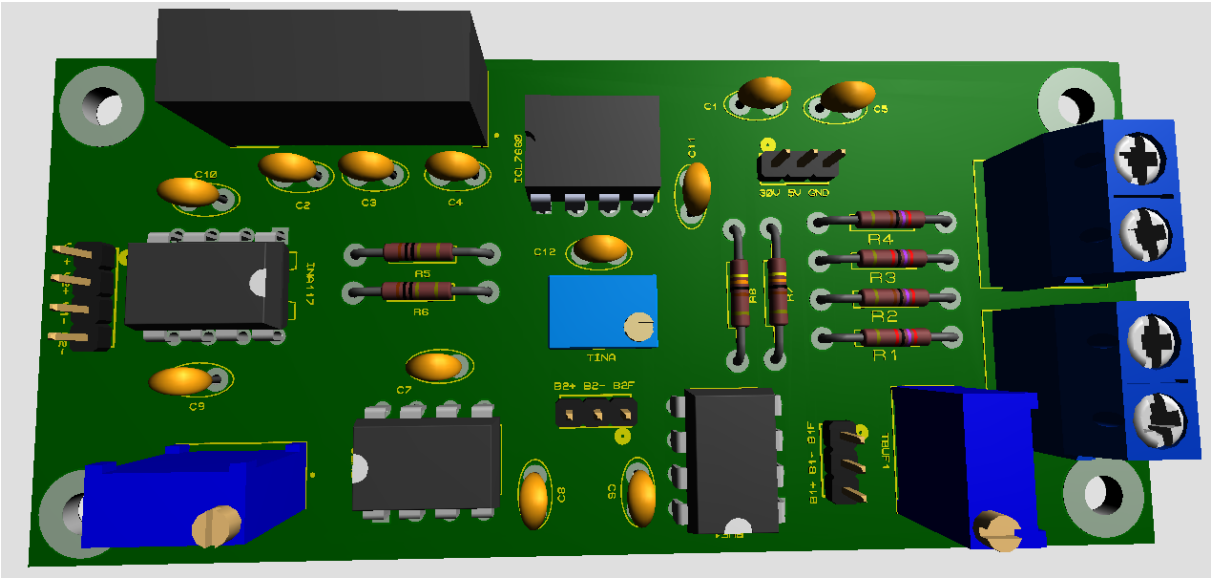


Figura 36 – Modelo 3D do projeto final na placa.

Fonte: Produzida pelo autor

## 5.5 Módulo de *Hardware*: *hardware.py*

Com a divisão mencionada na Seção 5.2, já é possível discutir o funcionamento de cada módulo que realiza a integração do software. O módulo de *hardware* é responsável por realizar a comunicação em mais "baixo nível" (termo utilizado porque não é uma comunicação em baixo nível no sentido literal, mas sim no sentido mais prático, já que todas as partes do programa já são suficientemente abstratas) do sistema. Por exemplo, o módulo de *hardware* controla a comunicação em *Modbus* e *1-wire*, que compõem a maior parte dos sensores. As funções principais incluem:

- *devices_read*: lê um conjunto de informações sobre sensores, como nome, id em uma rede, registradores, parâmetros associados a registradores, entre outros;
- *devices_load*: carrega as informações de devices_read para uma lista em memória, onde é possível gerenciar as informações de cada sensor por meio do acesso de uma lista dentro do código (consequentemente abstraindo cada dispositivo a uma simples entrada em um dicionário na linguagem Python);
- *master_routine*: responsável por fazer a leitura dos sensores Modbus e *1-wire* e retornar erros caso haja problemas na leitura.

Variantes das funções mencionadas acima também estão disponíveis dentro do código, como *owire_load* e *owire_data*, que realizam tarefas parecidas com *devices_read*. Porém o que é de essencial entendimento nesse módulo é que o mesmo realiza a comunicação **apenas** com os sensores da rede Modbus e *1-wire*, além de algumas funções miscelânea para tratamento de parâmetros de sensores, carregados no vetor de sensores, além de inicialização dos dispositivos. Um diagrama que descreve melhor o funcionamento desse módulo pode ser observado na Figura 37.

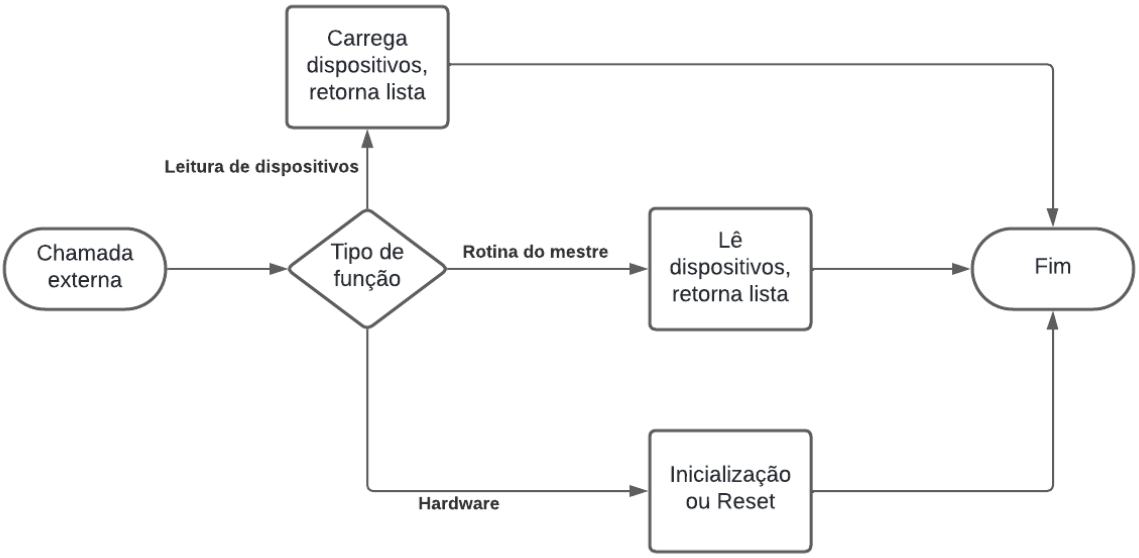



Figura 37 – Fluxograma para o módulo hardware.py.

Fonte: Produzida pelo autor

## 5.6 Módulo SenseHat/SPI: *shspi.py*

Esse módulo é responsável exclusivamente pela comunicação com o módulo *SenseHat* do sistema e com os conversores AD instalados, que trazem as informações do circuito integrador analógico e dos sensores de corrente instalados. Como as grandezas vindas desses dispositivos precisam ser amostradas em alta taxa (5 amostras por segundo), não há outras funções além das funções que coletam as grandezas dos conversores e do módulo *SenseHat*. Funções de inicialização de parâmetros também estão presentes (como a função *initialize*). Por fim, é esse módulo que define os fatores de conversão utilizados (lidos de um arquivo JSON) e realiza a conversão para obter o valor verdadeiro de alguma grandeza após a mesma ter passado por processamento analógico, como também inicializa as bibliotecas de comunicação com os pinos GPIO do Raspberry Pi, o *driver* de comunicação AD7705, além de uma função extra para separar a coleta em alta taxa em quatro arquivos, um para cada parte do dia, pois esses arquivos possuem muitas linhas de dados. Como cada um desses módulos só possuem funções de inicialização e funções de leitura que permanecem em laço de repetição infinito, seus fluxogramas estão contidos na Figura 40.

## 5.7 Módulo de Dados: *datapath.py*

O módulo de dados é um módulo auxiliar para leitura, escrita e *upload* de arquivos. É nesse módulo que é realizada a escrita de grandezas coletadas de todos os sensores em arquivos *csv*, a remoção de sensores da rede em caso de falhas constantes e o *upload* dos

arquivos para o *Google Drive*, além da escrita no arquivo *log* do sistema, arquivo responsável por registrar os eventos enviados pelo sistema (como o *reset* da rede modbus em caso de falha de um sensor, upload de arquivos, reinicialização do sistema, entre outros). O fluxograma da Figura 38 demonstra o funcionamento desse módulo.

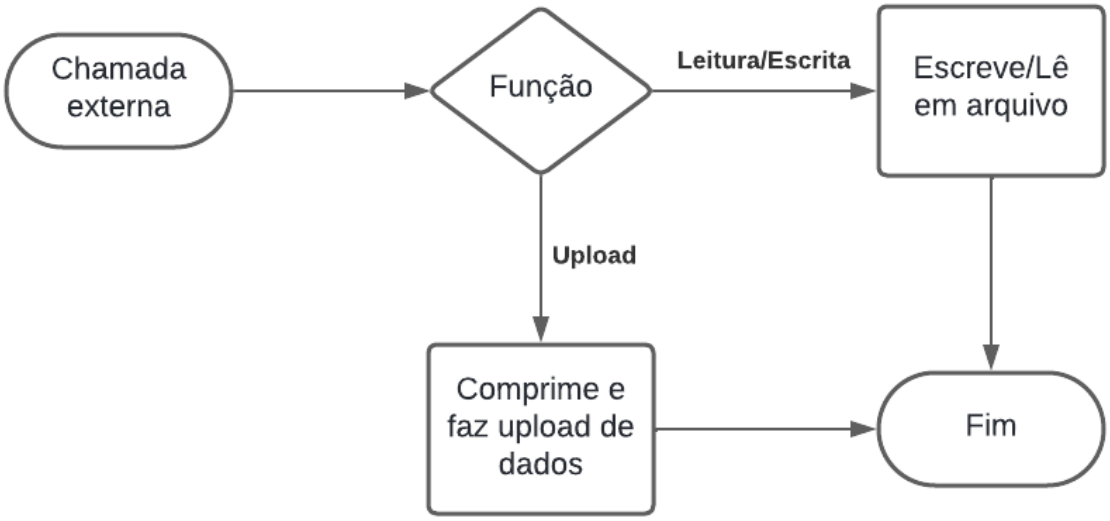



Figura 38 – Fluxograma para o módulo datapath.py.

Fonte: Produzida pelo autor

## 5.8 Módulo GUI: *gui.py*

Esse módulo é responsável pela implementação de uma interface gráfica (GUI), responsável por facilitar a visualização dos dados coletados em baixa taxa de amostragem, além de algumas informações que são demonstradas em uma caixa de mensagem no canto inferior direito da tela do usuário. É uma implementação simples, com uma lógica de programação não muito complicada. A interface não apresenta todas as informações presentes no sistema e é apenas um componente adicional, já que a visualização de tantos dados no terminal não é conveniente. Dessa forma, ambos GUI como terminal são utilizados como monitores de sistema.

## 5.9 Módulo Principal: *main.py*

A função do módulo principal é a coordenação da ordem dos eventos no sistema. É responsável por fazer a inicialização do núcleo do sistema (isto é, trazer todas as informações mais rudimentares para a área de memória onde o programa está executando, como quais são os sensores que serão lidos, onde serão armazenados dentro do código, entre outros).

Além disso, dentro desse módulo está contido o laço de repetição em baixa frequência e a inicialização da *thread* que mostrará a interface gráfica. Esse módulo também é responsável por fazer o *cleanup* do sistema quando o mesmo é desligado (fechar as bibliotecas de GPIO para que não haja erros na próxima execução do programa). A reinicialização não precisa ser "limpa", já que o sistema ao reiniciar possui uma instância sem lixo de memória ou processos associados ao programa. A Figura 39 demonstra um funcionamento simplificado desse módulo.

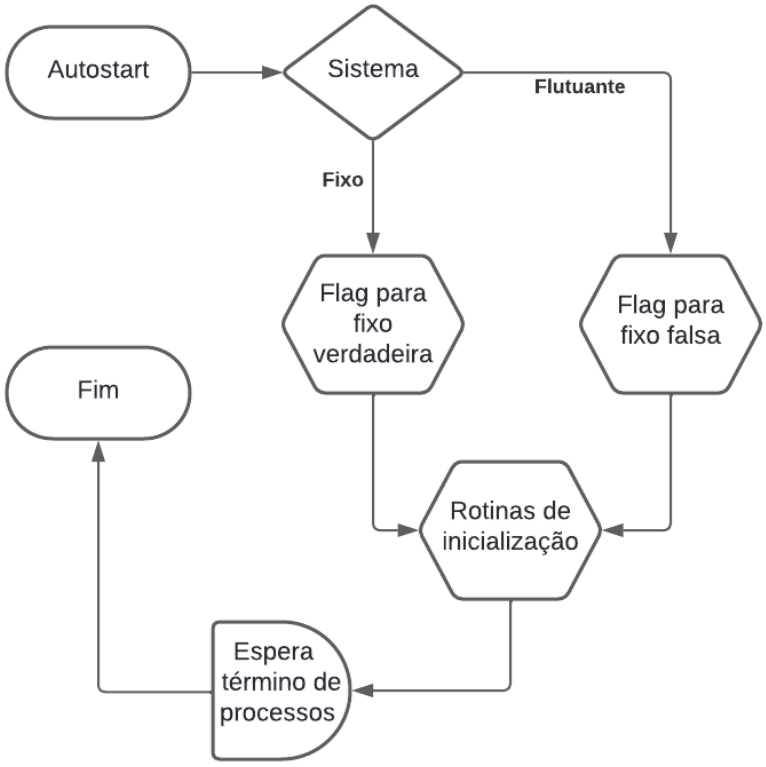



Figura 39 – Fluxograma para o módulo main.py.

Fonte: Produzida pelo autor

O módulo principal é responsável também por "lançar" os processos do sistema, fazendo com que componentes separados executem em paralelo. Dessa forma, uma visão geral do funcionamento do sistema pode ser observada na Figura 40.

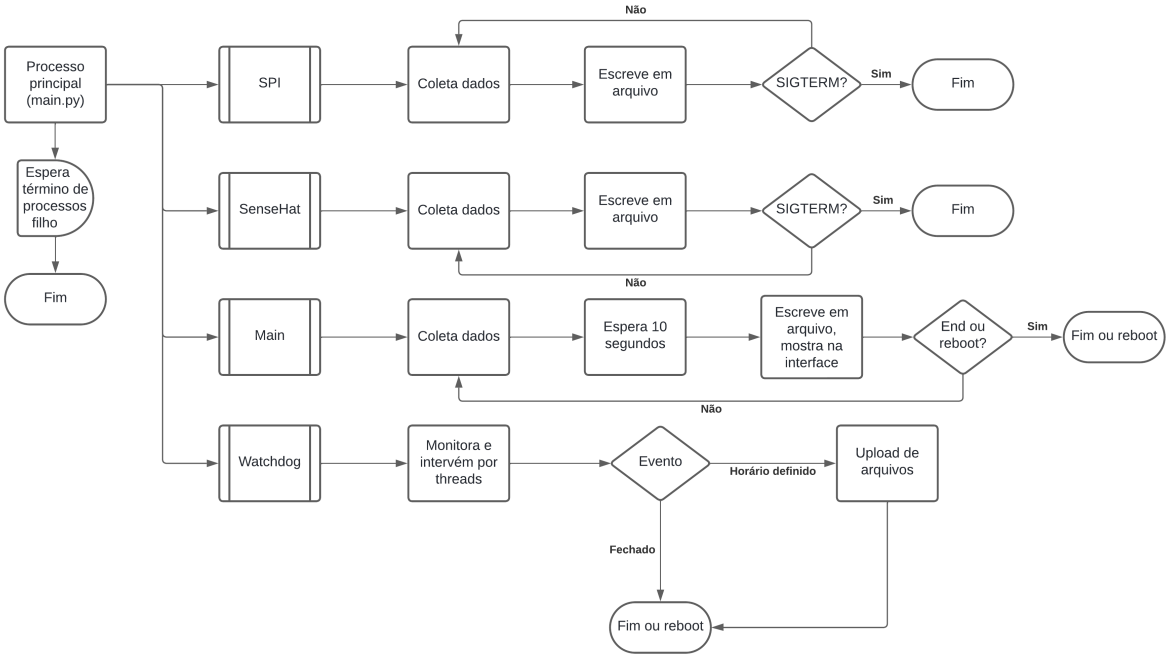



Figura 40 – Fluxograma para o sistema completo.

Fonte: Produzida pelo autor

## 5.10 Módulo *Watchdog*: *watchdog.py*

O último módulo *Watchdog*, responsável por prover o gerenciamento necessário para o sistema e executar ações de manutenção, tem funções implementadas de extrema importância como:

- Reiniciar o sistema em determinado horário;
- Monitorar os sensores a cada leitura;
- Aplicar um *reset* na rede Modbus em caso de falhas repetitivas de sensor;
- Remover um sensor da rede sob certas condições de falha (atualmente, após 3 *resets* na rede devido a esse sensor);
- Ligar luzes de sinalização à noite e desligá-las de dia;
- Disparar ventoinhas de refrigeração em determinada temperatura, e desligá-las quando a temperatura não for prejudicial aos componentes eletrônicos;
- Disparar o upload dos dados a partir de certo horário;
- Outras funções úteis para manutenção do sistema.

A Figura 41 demonstra um fluxograma que descreve o funcionamento desse módulo no sistema.

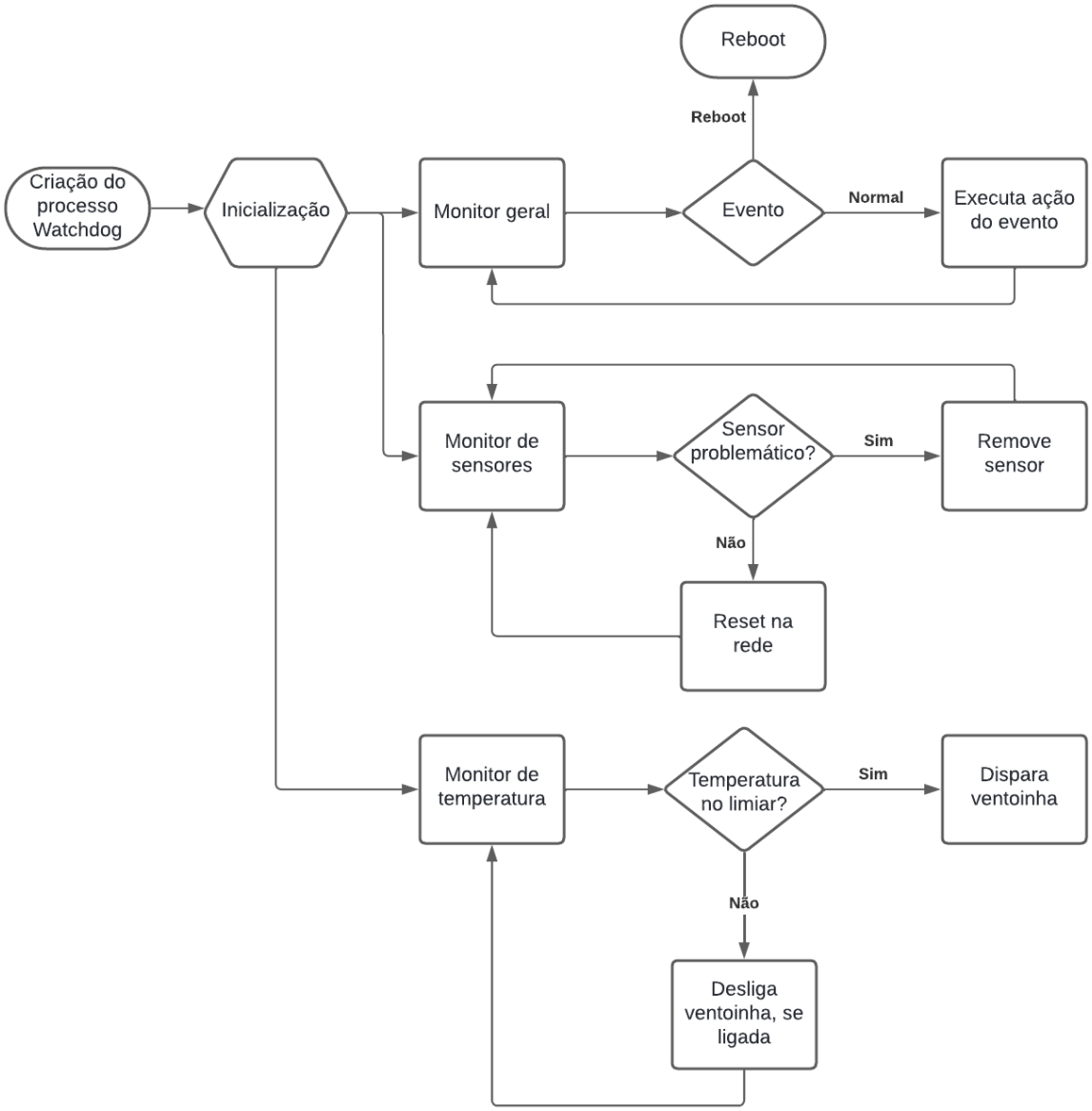



Figura 41 – Fluxograma para o módulo watchdog.py.

Fonte: Produzida pelo autor

## 5.11 Computação paralela

O *software* de integração, devido a seus diversos requisitos, exige uma complexidade muito maior do que simplesmente a execução de um *script* em Python. Apesar da linguagem Python ser bem simples e versátil, os problemas solucionados na implementação desse *software* envolvem conceitos avançados de ciência da computação. Antes de enunciá-los, é interessante rever os objetivos do *software* de automação:

- Integrar os sensores das redes Modbus e *1-wire*;
- Integrar os módulos *SenseHat* e SPI;
- Leitura simultânea de grandezas vindas dos sensores Modbus e *1-wire* em baixa frequência, e conjunto SPI e *SenseHat* em alta frequência (mínimo 2 amostras por segundo);
- Upload periódico (diário) dos dados obtidos em um dia;
- Gerenciamento e manutenção dos sistemas.

Pelos requisitos apresentados acima, pode-se observar que há muitas tarefas a serem executadas "ao mesmo tempo". Vale mencionar que além de todos esses requisitos, a interface gráfica não pode impedir a execução do programa como um todo. A grande quantidade de tarefas requer o uso simultâneo de concorrência (*threads*) e paralelismo (processos). Além disso, o módulo *Watchdog*, em hipótese alguma, pode deixar de ser executado, o que também requer uma independência do resto do código. A única solução para esse problema é o processamento paralelo.

### 5.11.1 Processos

O sistema, ao executar as rotinas de inicialização e fazer todas as verificações necessárias, invoca instâncias dos seguintes processos, pelos seguintes motivos:

- Processo principal: responsável pelo laço de repetição de baixa frequência e interface gráfica;
- Processo SPI: coleta em alta taxa do conjunto SPI;
- Processo SenseHat: coleta em alta taxa do módulo SenseHat;
- Processo *Watchdog*: execução de forma independente do módulo *Watchdog*.

Cada instância de um processo, ao ser iniciada, realiza uma operação conhecida como *fork*. Tal operação é uma chamada de sistema dos sistemas baseados em *Unix* (como o Linux), onde cada processo ao ser instanciado, recebe uma cópia do código fonte e todas as variáveis presentes no programa até a execução no momento em que é instanciado. Esse conjunto de variáveis é independente para cada processo, o que significa que a mesma variável com o mesmo nome no processo A pode e muitas vezes será diferente dessa variável no processo B. Isso traz vantagens e desvantagens, mas a principal desvantagem nesse escopo é que essa independência é indesejada, justamente pelo código de cada implementação já ser projetado para impedir conflitos. A solução desse problema é justamente a utilização de mecanismo de *intercomunicação entre processos*, um conceito fundamental de paralelismo em computação.

### 5.11.2 Intercomunicação de processos

Uma das formas mais fáceis de compartilhar variáveis entre dois processos é através de filas. Filas são regiões na memória gerenciadas pelo sistema operacional, onde cada processo pode colocar um objeto, que por sua vez pode ser coletado por outros processos. A principal vantagem dessa abordagem é que não é preciso conhecer o tamanho em bytes da estrutura, basta colocar o elemento e retirá-lo em uma instância de outro processo, desde que esses processos estejam relacionados de alguma forma. Uma desvantagem é que os elementos são retirados na ordem em que foram colocados, o que nem sempre é interessante, pois às vezes deseja-se elementos que foram colocados por último na fila, mas não é possível retirá-los sem antes retirar os elementos que foram enfileirados anteriormente. Isso requer criação de mais filas e gerenciamento das mesmas, o que complica um pouco o projeto do programa como um todo. Apesar disso, essa abordagem foi escolhida por realmente ser a de mais fácil implementação atualmente, mesmo não sendo a melhor solução. A Figura 42 demonstra como as filas são utilizadas dentro do programa para trocar variáveis entre os processos. As filas são separadas por nome e número, sendo:

- Fila 1 - *Watchdog*: fila utilizável pelo módulo *Watchdog*, para receber dados de outros programas;
- Fila 2 - Fila principal: uso geral, dados com certa importância são colocados nessa fila;
- Fila 3 - Fila de exceções: fila utilizada para sinalizar problemas a outros programas, que baseado na exceção enviada por essa fila executará algum tipo de ação (como recarregar os dispositivos na lista de leitura quando um sensor é removido da rede);
- Fila 4 - Fila *SenseHat*: fila de dados para o processo de coleta *SenseHat*;
- Fila 5 - Fila SPI: fila de dados para o processo de coleta *SPI*.

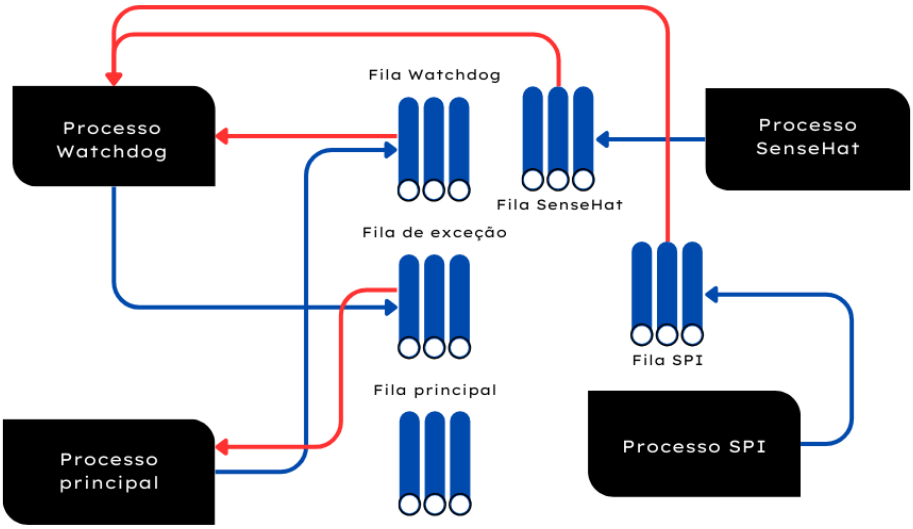



Figura 42 – Diagrama de filas utilizadas para comunicação de processos.

Fonte: Produzida pelo autor

Além disso, certas funções fornecidas por essas filas possibilitam operações blocantes (ou seja, operações que esperam por algum evento para continuar a execução), o que facilita muito o tratamento de exceções no programa ou tarefas que envolvem condições de corrida (condição em que a ordem de execução das linhas de algum programa interfere na computação desejada). A Figura 42 é um esquemático de como as filas são utilizadas pelos diferentes processos para transmitir dados de um processo para outro. As setas vermelhas indicam um dado que foi enfileirado e está sendo recebido por outro processo, já as setas azuis indicam enfileiramento de um dado por um processo. A fila principal, até o momento em que esse material foi escrito, não estava sendo usada. É possível observar um caminho fechado entre os processos *Watchdog* e principal (processo principal coloca dado na fila watchdog, processo *watchdog* recebe o dado da fila e insere outro dado para o processo principal na fila principal, terminando o ciclo com o processo principal recebendo esse dado). Os processos *SenseHat* e SPI apenas enfileiram dados até o momento.

# 6 Análise de Resultados

Após a integração de todos os barramentos de comunicação e implementação da camada de gerenciamento do sistema, é possível discutir os detalhes de seu funcionamento. Para saber se o resultado da integração foi satisfatório, é pertinente relembrar os objetivos mencionados no início desse trabalho. No que diz respeito ao software de automação, o mesmo deve ser capaz de realizar a coleta e envio de dados de forma completamente automática (isto é, sem intervenção humana) de todos os barramentos (I2C, SPI, Modbus) presentes no sistema. Além dos requisitos funcionais, é interessante que o sistema seja capaz de intervir em si mesmo na ocorrência de certos "eventos", isto é, na presença de acontecimentos que possam afetar o sistema de uma forma negativa. Logo, os aspectos a se considerarem são:

- O sistema realiza a coleta de dados corretamente?
- O sistema requer alguma forma de intervenção humana? Se sim, a partir de qual ponto?
- O sistema é gerenciado corretamente?
- A integração realizada possui limitações? Quais?

Considerando essas questões, alguns testes foram realizados e seus resultados foram coletados. A seguir são documentadas considerações sobre o sistema, como também os testes realizados. Os códigos dos módulos do software de automação podem ser encontrados no Apêndice B. É importante frisar que os códigos ali listados não são suficientes para o funcionamento do software, pois ele necessita de parâmetros de configuração tanto do *Google Drive* como dos sensores. Portanto, os Códigos 11 a 15 são apenas para fim de referência, devendo o leitor consultar o repositório listado no mesmo apêndice para mais informações.

## 6.1 Aspectos funcionais

Os primeiros testes consistem justamente na execução do programa, para verificar se o programa está correto logicamente. O programa foi então executado durante o dia inteiro, todos os dias, por cerca de 21 dias. Alguns desses dias com pequenas manutenções e alterações, outros sem nenhum tipo de intervenção. O interesse se concentra nos dias em que não houve intervenção humana e que apresentaram algum tipo de funcionamento peculiar, em um cenário onde o sistema está em funcionamento em uma sala que não possui o clima tão variável (o laboratório de desenvolvimento).

### 6.1.1 Execução e interface

A execução do programa ocorre toda vez que o sistema é inicializado, por meio de um *trigger* de inicialização configurado no próprio sistema operacional do *Raspberry Pi*. A interface do programa pode ser observada na Figura 43, em que é possível observar o laço de repetição em baixa frequência em *waiting* para atualizar as informações no *dashboard* do programa, na pequena tela de avisos no canto inferior direito do *software*. Após o tempo de espera, é possível ver as informações do barramento Modbus presentes na interface gráfica (Figura 44), e as informações do barramento SPI e I2C mostradas no terminal, como também variáveis que demonstram o estado do sistema naquele momento.

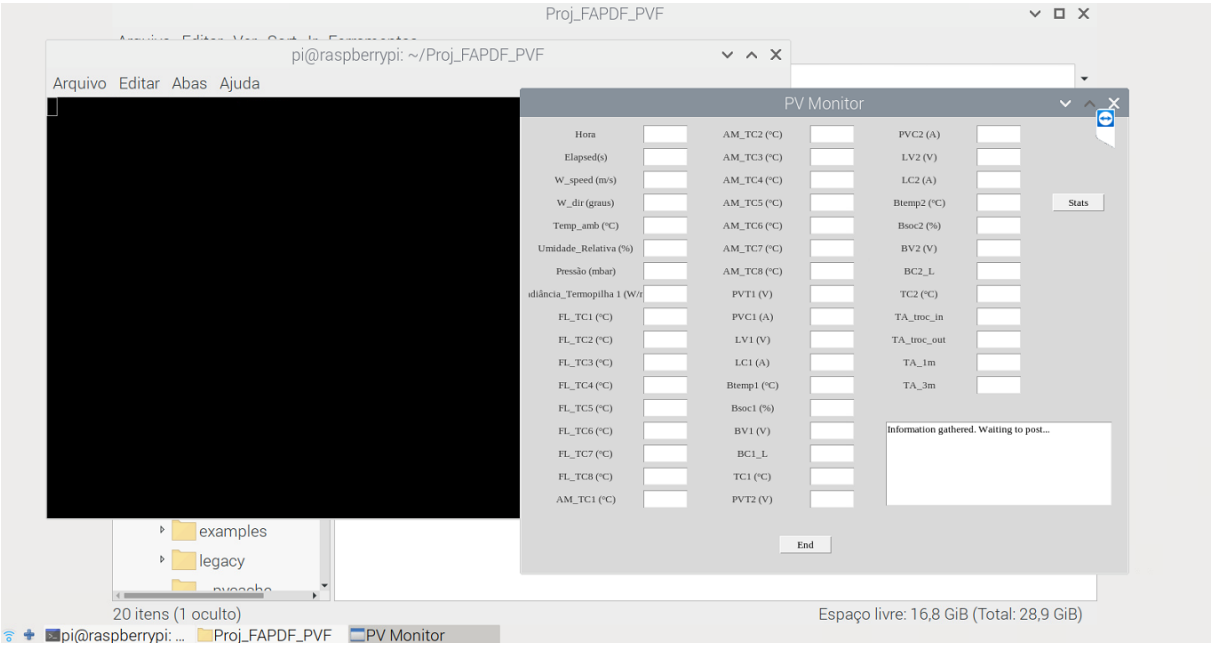



Figura 43 – Tela inicial do software de automação.

Fonte: Produzida pelo autor

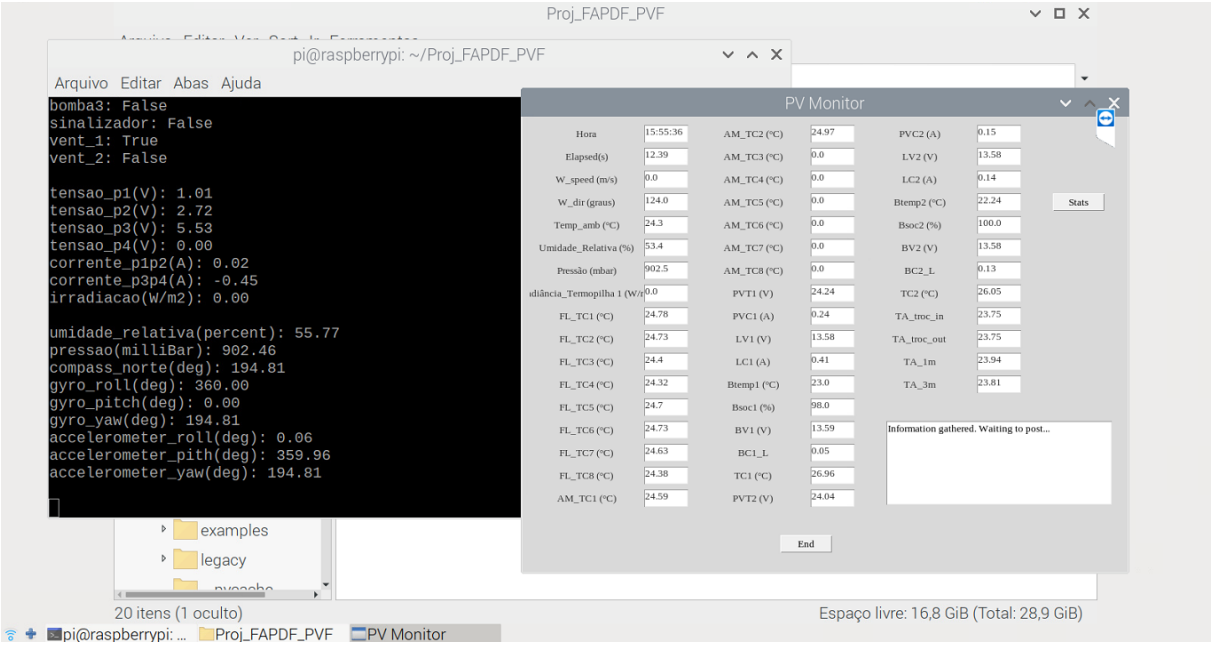



Figura 44 – Informações adquiridas dos sensores e mostradas na tela.

Fonte: Produzida pelo autor

Uma variável com o valor *False* indica que um componente responsável por algo no sistema está desligado, e uma variável com o valor *True* indica que certo componente está ativo. Como o teste da Figura 44 foi realizado às 15:55 do dia 31 de Maio, no Laboratório de Energia Solar da Faculdade de Tecnologia, onde os módulos não se encontravam em campo, apenas a ventoinha responsável pela refrigeração do *Raspberry Pi* foi acionada pelo sistema. Os demais componentes encontravam-se desligados, por não atenderem às condições de ativação.

Outra funcionalidade do programa é a capacidade de gerar curvas com as informações coletadas pelo sistema, lidas de um arquivo, sem interrupção da coleta em tempo real. Para isso, basta clicar no botão *Stats* e selecionar a grandeza coletada pelo sistema. É possível gerar mais de uma curva para comparação. A Figura 45 demonstra as curvas geradas pelo *software* de automação referentes ao Termopar 9, responsável pela medição da temperatura da face inferior do módulo inferior do sistema (que está próxima ao trocador de calor) e do cabo com um sensor que estará a 3 metros de profundidade dentro d'água em campo. No teste realizado, tal cabo estava registrando a temperatura ambiente da sala onde estavam os módulos fotovoltaicos. É possível concluir das curvas que a temperatura da sala é menor por volta das 6 horas da manhã, e que a temperatura da face do módulo inferior tem uma diferença de cerca de 1 °C em relação ao cabo de 3 metros. Essas curvas podem ser úteis para comparação entre diferentes grandezas para tirar conclusões parciais a respeito do módulo fotovoltaico em campo. É possível dar *zoom* e até mesmo salvar as curvas em um arquivo de imagem. Devido à alta quantidade de dados em alta frequência, optou-se pela não implementação de seu gerador de curvas. As curvas resultantes de dados coletados em

alta frequência podem ser traçadas por qualquer *software* gerador de gráficos, comercial ou não.

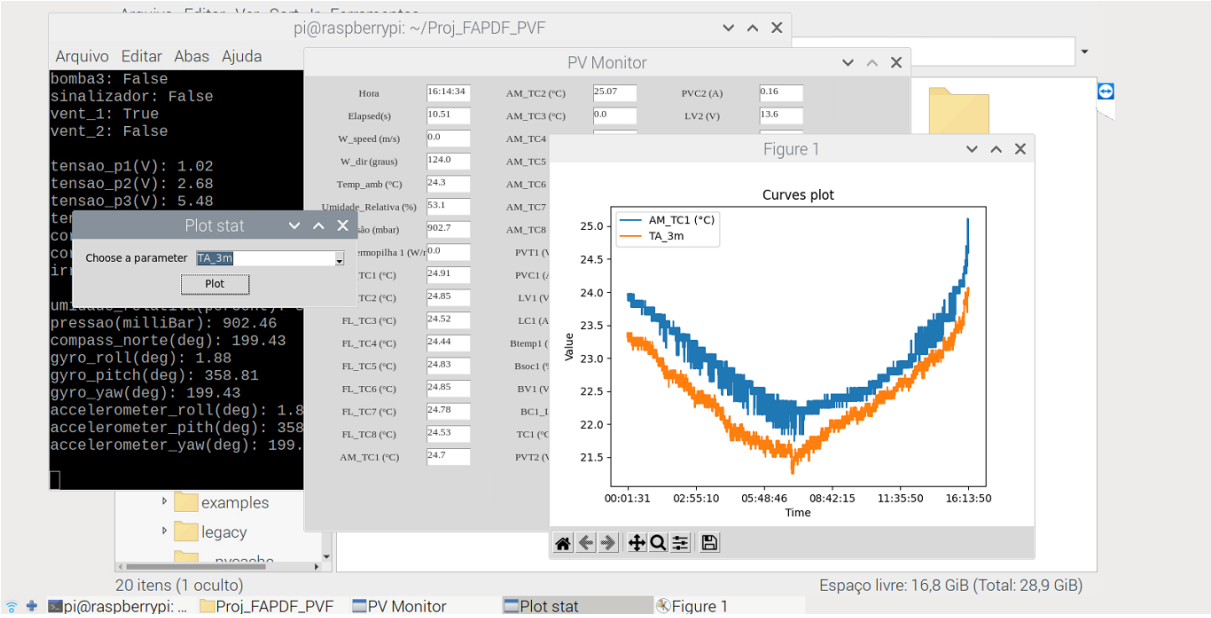



Figura 45 – Curvas geradas pelo *software* de automação.

Fonte: Produzida pelo autor

Ao clicar no botão *End*, a execução do programa é finalizada, juntamente com a coleta de dados de todos os barramentos. É possível continuar utilizando o sistema independente da execução do programa, porém uma leitura de um sensor por outro programa quando o barramento estiver ocupado ocasionará um erro, portanto é recomendado parar a coleta antes. Caso o usuário não pare a coleta manualmente, a coleta só se encerra na rotina de salvamento e *reboot* do sistema, que acontece das 23h até 0h, horário em que é iniciada novamente.

### 6.1.2 Salvamento de dados e envio à nuvem

Os dados coletados pelo programa são em sua maioria salvos em arquivos .csv (Figura 46), visualizáveis por algum programa capaz de abrir e processar dados em planilhas, para posterior utilização em softwares como *Excel*, por exemplo. Os únicos dados que não são salvos nesse formato é o arquivo *log* do sistema, responsável por informar eventos pertinentes para eventual necessidade de depuração ou de informações úteis para análise. Esses arquivos são comprimidos em um único arquivo .zip (arquivo compactado), que contém o diretório de dados em alta frequência, o arquivo *csv* de dados em baixa frequência e o registro de eventos do sistema. Esse arquivo é enviado ao *Google Drive*, onde é possível o acesso por diferentes dispositivos. A rotina de envio é acionada pelo módulo *Watchdog* às 23h, todos os dias. A Figura 47 demonstra os arquivos compactados gerados pelo sistema. Tais arquivos podem

ser vistos já carregados no *Google Drive* na Figura 48. Por fim, após efetuar o *download* dos arquivos do dia 26 de Maio de 2024, por exemplo, é possível acessar o conteúdo do arquivo de qualquer máquina capaz de abrir arquivos formato .csv, nesse caso, o computador pessoal do autor, como demonstrado na Figura 49.

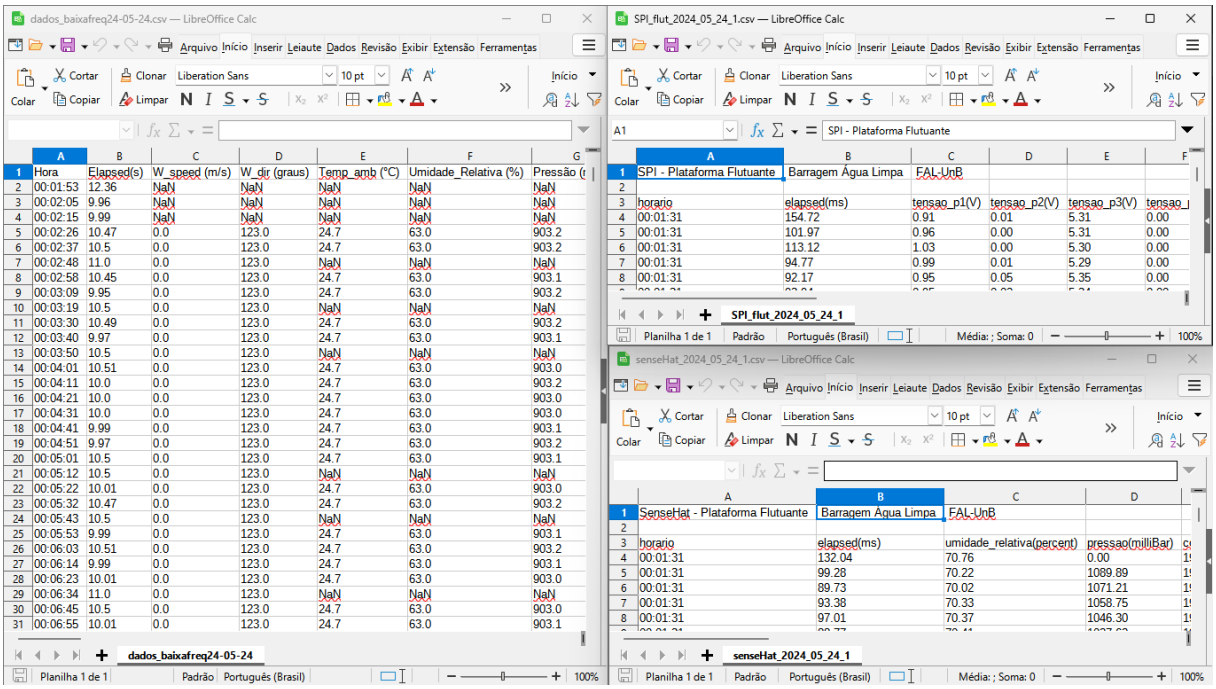



Figura 46 – Planilhas csv dos barramentos.

Fonte: Produzida pelo autor

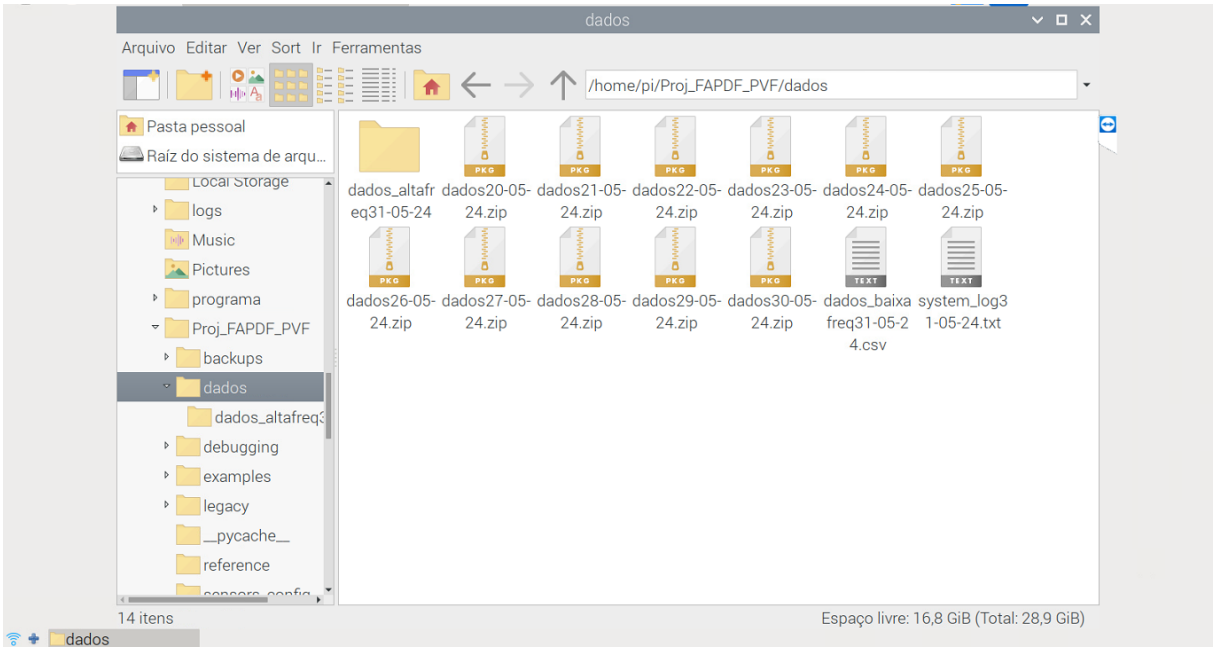



Figura 47 – Arquivos compactados gerados pelo sistema.

Fonte: Produzida pelo autor

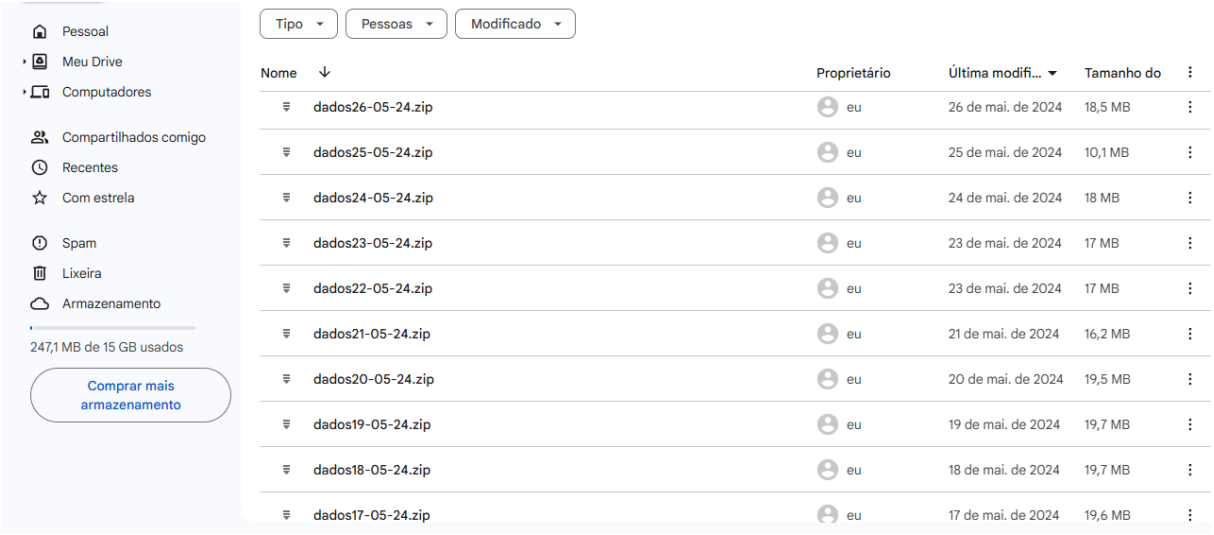



Figura 48 – Arquivos compactados carregados no *Google Drive*.

Fonte: Produzida pelo autor

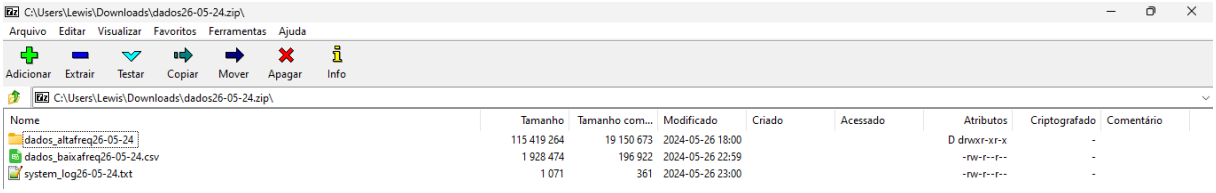



Figura 49 – Conteúdos do arquivo compactado.

Fonte: Produzida pelo autor

### 6.1.3 Configuração

Uma das vantagens da implementação aqui apresentada é a possibilidade de adicionar, remover e configurar sensores no mesmo *software* de automação, pois o mesmo escolhe automaticamente os parâmetros para cada sistema, basta utilizar uma *flag* em sua execução indicando em qual tipo de sistema o programa está em execução, o que também pode ser configurado no sistema do próprio *Raspberry Pi*.

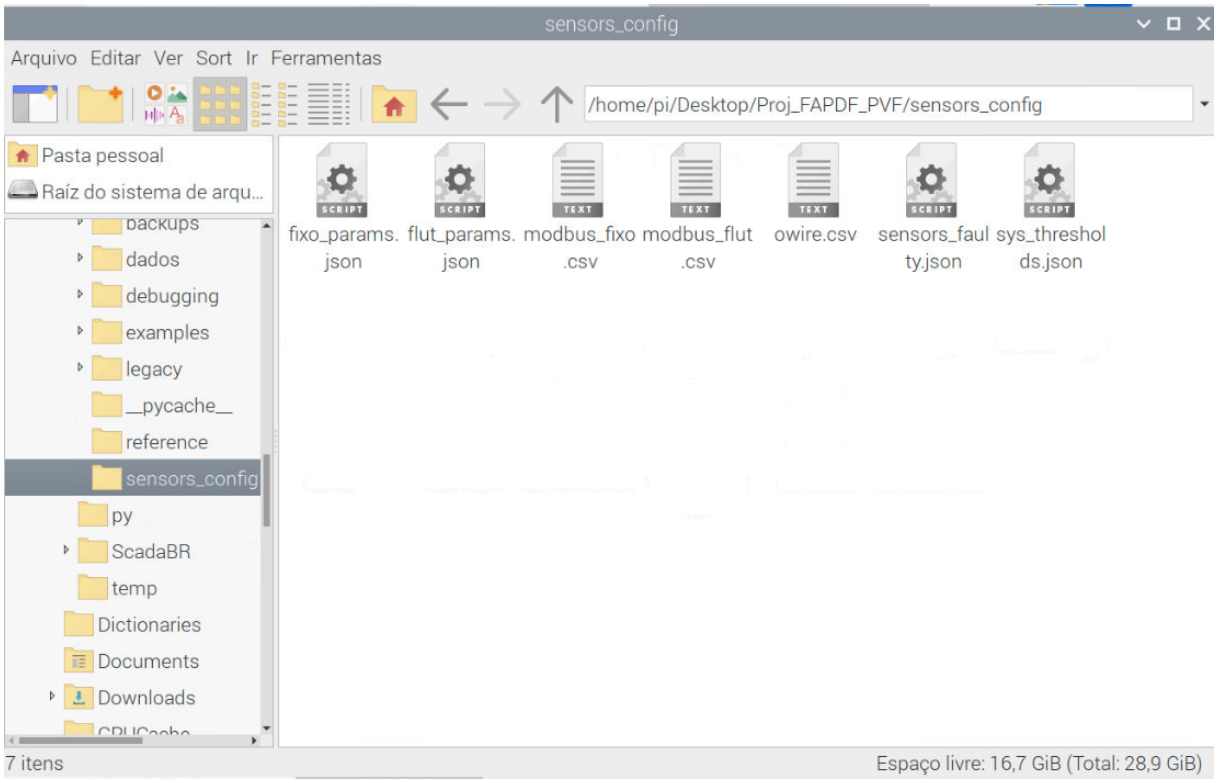



Figura 50 – Arquivos de configuração do sistema.

Fonte: Produzida pelo autor

Os arquivos demonstrados na Figura 50 consistem nos arquivos de configuração utilizados pelo sistema em diferentes ocasiões, o sistema escolhe o arquivo correto e sabe quais não são utilizados para cada sistema automaticamente, sendo esses:

- *fixo_params.json*: parâmetros do conjunto SPI para o sistema fixo;
- *flut_params.json*: parâmetros do conjunto SPI para o sistema flutuante;
- *sys_thresholds.json*: parâmetros de limites para as condições de acionamento e desligamento do sistema;
- *sensors_faulty.json*: arquivo detector de sensor inativo. Utilizado para remoção e redetecção em caso de eventual resposta;
- *modbus_fixo.csv*: parâmetros de sensores Modbus do sistema fixo;
- *modbus_flut.csv*: parâmetros de sensores Modbus do sistema flutuante;
- *owire.csv*: parâmetros de sensores *1-wire* do sistema flutuante.

As Figuras 51 e 52 demonstram quais parâmetros são esses e como eles são configurados dentro dos arquivos. Por exemplo, o sistema ao fazer as requisições Modbus aos sensores

consegue identificar os parâmetros de baixo-nível (quais sensores ler, qual o endereço do dispositivo na rede, quais registradores ler) e realizar a conversão das grandezas automaticamente através desse arquivo, idem para os sensores *1-wire*. Os arquivos json tornam de fácil configuração os parâmetros elétricos (dos sensores do conjunto SPI) e alguns pormenores como o horário de inicialização do sistema, a temperatura de ativação da ventoinha (atualmente configurada para ativação caso a temperatura do componente seja maior que 48 °C), entre outros parâmetros relevantes, sem necessidade de alteração no código-fonte.

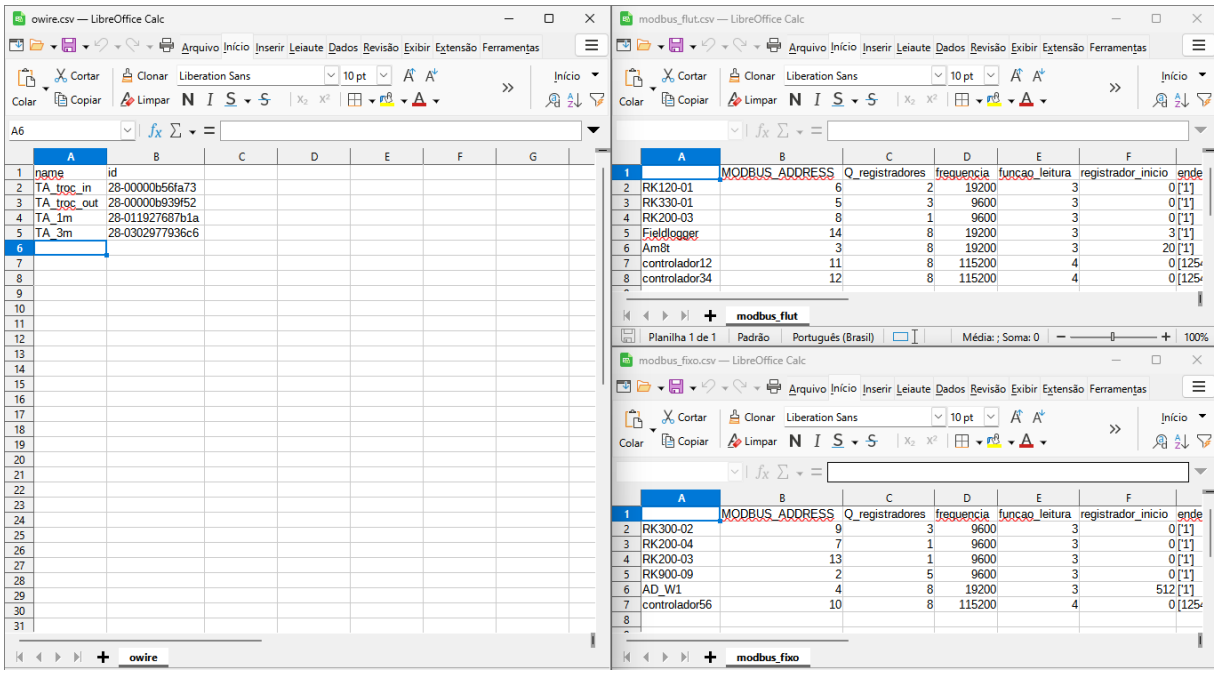



Figura 51 – Arquivos de configuração do sistema - Modbus.

Fonte: Produzida pelo autor

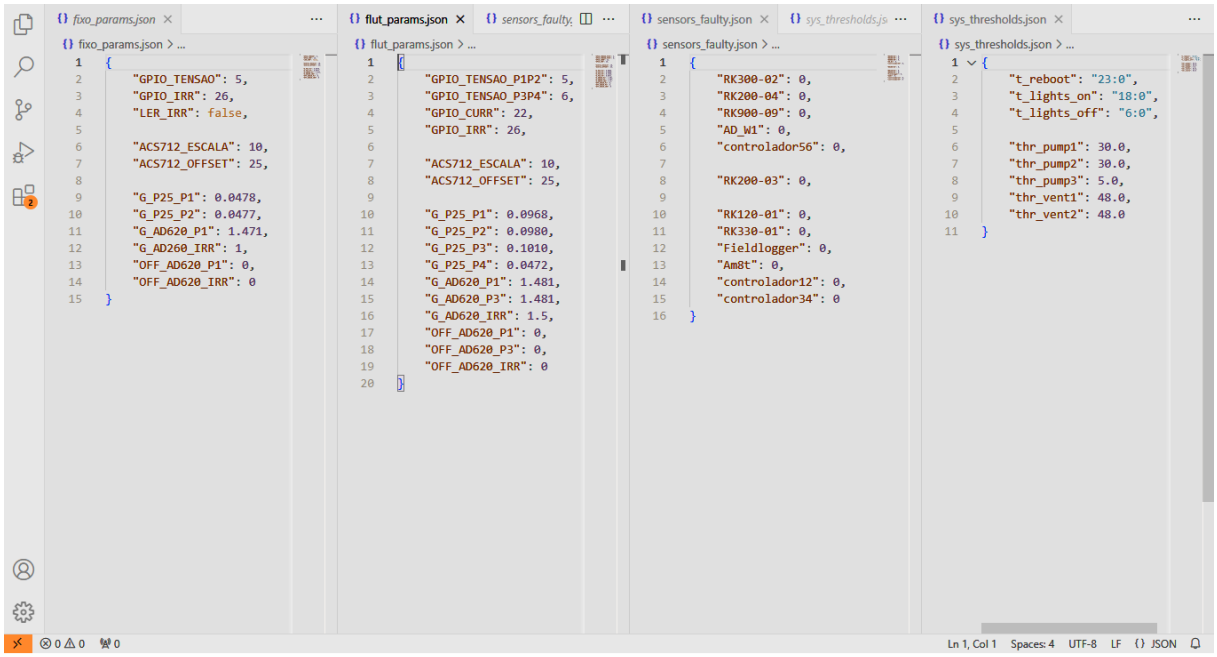



Figura 52 – Arquivos de configuração do sistema - JSON.

Fonte: Produzida pelo autor

## 6.2 Camada de gerenciamento

Como mencionado anteriormente, a camada de gerenciamento do sistema é implementada pelo módulo *Watchdog*. Os principais aspectos a se investigar é se essa camada (ou processo em execução) consegue intervir diretamente no sistema, de modo a tentar manter seu funcionamento correto. O sistema inclui uma rotina de registro de eventos (*logging*), que guarda os detalhes importantes em um arquivo de texto. Para análise, foram colhidos quatro arquivos de registro onde o sistema estava em execução durante todo o dia, tais registros podem ser observados nos Códigos 1, 2 e 3.

Código 1 – Registro do dia 25 de Maio

```
[00:00:30] - O sistema foi limpo.
[00:01:47] - O sistema foi iniciado com sucesso.
[00:01:47] - O módulo de watchdog foi iniciado e está monitorando o sistema.
[00:01:51] - A temperatura da CPU é de 58.913 graus. O ventilador está agora ligado.
[00:01:57] - As luzes de sinalização foram acesas.
[07:00:06] - As luzes de sinalização foram apagadas.
[08:00:23] - Recarregando Modbus devido a sensor com defeito: RK330-01
[18:00:01] - As luzes de sinalização foram acesas.
[23:00:06] - O sistema está agora em modo de espera. Nenhum dado está sendo monitorado.
[23:00:06] - O sistema reiniciará em 1 hora.
[23:00:06] - A rotina de upload foi iniciada.
```

Código 2 – Registro do dia 29 de Maio

```
[00:00:21] - O sistema foi limpo.
[00:01:09] - O sistema foi iniciado com sucesso.
[00:01:09] - O módulo de watchdog foi iniciado e está monitorando o sistema.
[00:01:13] - A temperatura da CPU é de 58.426 graus. O ventilador está agora ligado.
[00:01:35] - As luzes de sinalização foram acesas.
```

```
[07:00:00] - As luzes de sinalização foram apagadas.
[18:00:02] - As luzes de sinalização foram acesas.
[23:00:06] - O sistema está agora em modo de espera. Nenhum dado está sendo monitorado.
[23:00:06] - O sistema reiniciará em 1 hora.
[23:00:06] - A rotina de upload foi iniciada.
```

Código 3 – Registro do dia 1 de Junho

```
[00:00:24] - O sistema foi limpo.
[00:01:24] - O sistema foi iniciado com sucesso.
[00:01:24] - O módulo de watchdog foi iniciado e está monitorando o sistema.
[00:01:28] - A temperatura da CPU é de 58.913 graus. O ventilador está agora ligado.
[00:01:34] - As luzes de sinalização foram acesas.
[00:02:14] - Recarregando Modbus devido a sensor com defeito: controlador12
[00:02:48] - A bomba da cortina foi ligada.
[00:02:58] - A bomba da cortina foi desligada.
[07:00:00] - As luzes de sinalização foram apagadas.
[18:00:01] - As luzes de sinalização foram acesas.
[23:00:01] - O sistema está agora em modo de espera. Nenhum dado está sendo monitorado.
[23:00:01] - O sistema reiniciará em 1 hora.
[23:00:01] - A rotina de upload foi iniciada.
```

Código 4 – Registro do dia 5 de Junho

```
[00:00:35] - O sistema foi limpo.
[00:01:40] - O sistema foi iniciado com sucesso.
[00:01:40] - O módulo de watchdog foi iniciado e está monitorando o sistema.
[00:01:43] - A temperatura da CPU é de 57.939 graus. O ventilador está agora ligado.
[00:01:50] - As luzes de sinalização foram acesas.
[07:00:01] - As luzes de sinalização foram apagadas.
[15:17:47] - Recarregando Modbus devido a sensor com defeito: RK330-01
[15:18:44] - Recarregando Modbus devido a sensor com defeito: RK330-01
[15:19:40] - Recarregando Modbus devido a sensor com defeito: RK330-01
[15:20:14] - A bomba do backsheet foi ligada.
[15:20:24] - A bomba do backsheet foi desligada.
[15:20:36] - Recarregando Modbus devido a sensor com defeito: RK330-01
[15:21:10] - A bomba do backsheet foi ligada.
[15:21:20] - A bomba do backsheet foi desligada.
[15:21:32] - O sensor RK330-01 falha repetidamente. Ele não será mais lido.
[15:21:39] - O sensor RK330-01 no endereço 5 não está respondendo e, portanto, não será
    lido.
[15:23:29] - O sistema foi limpo.
[15:23:47] - O sistema foi iniciado com sucesso.
[15:23:47] - O módulo de watchdog foi iniciado e está monitorando o sistema.
[15:23:50] - A temperatura da CPU é de 62.809 graus. O ventilador está agora ligado.
[18:00:01] - As luzes de sinalização foram acesas.
[23:00:03] - O sistema está agora em modo de espera. Nenhum dado está sendo monitorado.
[23:00:03] - O sistema reiniciará em 1 hora.
[23:00:03] - A rotina de upload foi iniciada.
```

É possível ver que no dia 25 de Maio (Código 1) houve repetidas falhas no sensor RK330-01 (sensor de temperatura, umidade e pressão), o que foi detectado pelo sistema e corrigido com um *reset* da rede Modbus. É possível ver esse fato na Figura 53, tanto a parte da falha do sensor como o mesmo voltando a funcionar.

| 1 | Hora | Elapsed(s) | W_speed (m/s) | W_dir (graus) | Temp_amb (°C) | Umidade_Relativa (%) | Pressão (mbar) |
|---|---|---|---|---|---|---|---|
| 2819 | 07:59:29 | 9.97 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2820 | 07:59:40 | 10.49 | 0.0 | 124.0 | NaN | NaN | NaN |
| 2821 | 07:59:50 | 10.46 | 0.0 | 124.0 | NaN | NaN | NaN |
| 2822 | 08:00:01 | 11.0 | 0.0 | 124.0 | NaN | NaN | NaN |
| 2823 | 08:00:12 | 10.96 | 0.0 | 124.0 | NaN | NaN | NaN |
| 2824 | 08:00:23 | 15.44 | 0.0 | 124.0 | NaN | NaN | NaN |
| 2825 | 08:00:39 | 11.5 | 0.0 | 124.0 | 23.1 | 61.9 | 903.5 |
| 2826 | 08:00:49 | 9.99 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2827 | 08:00:59 | 9.99 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2828 | 08:01:09 | 10.49 | 0.0 | 124.0 | 23.1 | 61.9 | 903.3 |
| 2829 | 08:01:20 | 9.96 | 0.0 | 124.0 | 23.1 | 61.9 | 903.3 |
| 2830 | 08:01:29 | 9.96 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2831 | 08:01:40 | 10.0 | 0.0 | 124.0 | 23.1 | 61.9 | 903.3 |
| 2832 | 08:01:49 | 9.99 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2833 | 08:02:00 | 10.0 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2834 | 08:02:10 | 9.96 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2835 | 08:02:20 | 10.0 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2836 | 08:02:30 | 10.0 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2837 | 08:02:40 | 10.0 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2838 | 08:02:50 | 10.0 | 0.0 | 124.0 | 23.1 | 61.9 | 903.3 |
| 2839 | 08:02:59 | 9.97 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2840 | 08:03:10 | 9.99 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2841 | 08:03:20 | 9.94 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2842 | 08:03:30 | 9.99 | 0.0 | 124.0 | 23.1 | 61.9 | 903.3 |
| 2843 | 08:03:40 | 10.47 | 0.0 | 124.0 | NaN | NaN | NaN |
| 2844 | 08:03:50 | 10.01 | 0.0 | 124.0 | 23.1 | 61.9 | 903.3 |
| 2845 | 08:04:00 | 9.95 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2846 | 08:04:11 | 10.45 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2847 | 08:04:21 | 10.0 | 0.0 | 124.0 | 23.1 | 61.9 | 903.4 |
| 2848 | 08:04:31 | 10.0 | 0.0 | 124.0 | 23.1 | 61.9 | 903.3 |

Figura 53 – Correção de falha de sensor pelo módulo *Watchdog* - 25 de Maio.

Fonte: Produzida pelo autor

Outra ocasião onde é possível observar *reset* na rede devido a alguma falha de sensor é no dia 1 de Junho, onde o controlador de carga não respondia (Figura 54). Após a auto-correção do sistema, é possível observar que o sistema segue em funcionamento. Caso o sistema ainda assim não detectasse melhora, a rede seguiria sendo reinicializada até o sensor ser removido automaticamente da rede e sua leitura não mais realizada. Além disso, o Código 4 apresenta uma ocasião em que há falha do sensor RK330-01, em um teste feito manualmente pelo autor (fisicamente desconectando um sensor da rede), onde o sensor é removido da rede pelo sistema e recarregado logo em seguida. Reinserindo esse sensor na rede e recarregando o sistema, é possível observar que o mesmo segue funcionando, o que mostra que o sistema é capaz de identificar quando um sensor retorna à rede. Caso o RK330-01 ainda não retornasse resposta, a mensagem demonstrada ao iniciar o sistema seria: *O sensor RK330-01 no endereço 5 não está respondendo e, portanto, não será lido.*

| | A | X | Y | Z | AA | AB | AC | AD | AE | AF | AG | AH | AI | AJ | AK | AL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Hora | AM_TC8 (°C) | PVT1 (V) | PVC1 (A) | LV1 (V) | LC1 (A) | Btemp1 (°C) | Bsoc1 (%) | BV1 (V) | BC1_L | TC1 (°C) | PVT2 (V) | PVC2 (A) | LV2 (V) | LC2 (A) | Btemp2 (°C) |
| 2 | 00:01:31 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 3 | 00:01:43 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 4 | 00:01:53 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 24.04 | 24.04 | 24.04 | 24.04 | 24.04 |
| 5 | 00:02:03 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 24.04 | 24.04 | 24.04 | 24.04 | 24.04 |
| 6 | 00:02:13 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 24.04 | 24.04 | 24.04 | 24.04 | 24.04 |
| 7 | 00:02:30 | 0.0 | 24.3 | 0.24 | 13.57 | 0.39 | 22.47 | 98.0 | 13.57 | 0.09 | 26.84 | 24.04 | 0.18 | 13.59 | 0.17 | 22.31 |
| 8 | 00:02:40 | 0.0 | 24.29 | 0.23 | 13.6 | 0.33 | 22.54 | 100.0 | 13.6 | 0.07 | 26.75 | 24.05 | 0.17 | 13.58 | 0.16 | 22.3 |
| 9 | 00:02:50 | 0.0 | 24.31 | 0.22 | 13.6 | 0.33 | 22.47 | 100.0 | 13.6 | 0.08 | 26.82 | 24.03 | 0.16 | 13.6 | 0.17 | 22.32 |
| 10 | 00:03:00 | 0.0 | 24.3 | 0.23 | 13.63 | 0.31 | 22.49 | 100.0 | 13.63 | 0.08 | 26.83 | 24.02 | 0.18 | 13.59 | 0.22 | 22.32 |
| 11 | 00:03:10 | 0.0 | 24.28 | 0.22 | 13.61 | 0.31 | 22.46 | 100.0 | 13.64 | 0.06 | 26.86 | 24.04 | 0.16 | 13.58 | 0.17 | 22.32 |
| 12 | 00:03:20 | 0.0 | 24.3 | 0.22 | 13.61 | 0.29 | 22.47 | 100.0 | 13.61 | 0.06 | 26.87 | 24.02 | 0.18 | 13.55 | 0.17 | 22.31 |
| 13 | 00:03:30 | 0.0 | 24.31 | 0.22 | 13.61 | 0.34 | 22.47 | 100.0 | 13.64 | 0.07 | 26.84 | 24.05 | 0.17 | 13.58 | 0.17 | 22.22 |

Figura 54 – Correção de falha de sensor pelo módulo *Watchdog* - 1 de Junho.

Fonte: Produzida pelo autor

Note também que as rotinas responsáveis pela refrigeração do sistema, ativação e desativação de luzes de sinalização, *upload* de arquivos e reinicialização também funcionam adequadamente, como observado no registro do dia 29 de Maio (Código 2) e Figura 48. Ou seja, o sistema cumpre os requisitos necessários de funcionamento, sendo capaz de enviar todos os dados coletados periodicamente à nuvem, efetuar auto-correção em caso de falhas e monitorar a si mesmo, sem intervenção humana. Apesar disso, como todo sistema, apresenta limitações que serão discutidas posteriormente.

## 6.3 Limitações e vantagens

O *software* de automação apresenta como vantagem principal a sua versatilidade de utilização nos diferentes sistemas. De fato, é mais fácil e conveniente fazer alterações sem precisar fazer uma réplica do *software* para cada sistema, com códigos diferentes. Apesar de isso tornar a implementação um pouco mais complexa, o benefício na utilização é maior. Além disso, a escolha em salvar os arquivos em formato texto (que são altamente compressíveis) fornece uma economia de banda e espaço em disco altamente vantajosa. Para se ter uma noção do quão vantajoso isso é, pode-se realizar alguns cálculos simples. Suponha por exemplo que um arquivo não comprimido tenha tamanho em torno de 120 MB (atualmente o maior tamanho dá em torno de 111 MB) e um arquivo comprimido em torno de 20 MB (tamanho real está em torno de 18 MB). Logo, isso representa uma taxa de compressão de aproximadamente 83%! Sem a compressão, aproximadamente 3,6 GB de banda mensal (30 dias) seriam utilizados para envio de dados. Com a compressão, a utilização é reduzida para 600 MB. Por fim, tem-se que o plano gratuito do *Google Drive* fornece armazenamento em nuvem máximo de 15 GB. Caso houvesse monitoramento de dados por 1 ano sem interrupções, o espaço consumido seria de aproximadamente 7,4 GB (considerando 1 mês tendo 30 dias), o que possibilita o monitoramento por dois anos inteiros sem interrupções, o que torna um plano mensal de 8 GB de dados bem mais do que suficiente para envio de dados e operação remota via *TeamViewer* (uma solução de acesso remoto bastante conhecida). Ou seja, o sistema é eficiente em termos de armazenamento.

Apesar disso, há uma limitação notória de difícil correção no sistema automatizado. Se por algum motivo ocorrer um erro interno ao sistema (não relacionado ao *software* de automação) que venha a ocasionar perda de conexão com a internet, pode ocorrer de os dados no dia específico não serem enviados à nuvem, pois a rotina de *upload* falhará em perda de conexão. É claro que estamos nos referindo a um período prolongado de indisponibilidade, pois o sistema é reinicializado automaticamente todos os dias. Isso não é necessariamente um problema que acarretaria perda de dados, pois eles ficam salvos no cartão de memória do *Raspberry Pi*. O único cenário onde realmente haveria perda de dados é se por algum motivo a leitura desse cartão não pudesse ser feita por outro dispositivo.

Outro cenário seria em uma circunstância em que o *Raspberry Pi* ficasse em estado de *standby*, como no caso de acabar a bateria do sistema devido à falta de geração por ausência de irradiância solar. Esse cenário é de difícil ocorrência, pois geralmente o *Raspberry Pi* só entra nesse estado em um desligamento manual.

A questão é que os problemas "externos" ao sistema computacional dos módulos automatizados não têm solução simples. A identificação desses problemas pode ser ou não possível pela interface gráfica, desde que estejam relacionados aos sensores. O ideal seria que o sistema sempre fosse capaz de corrigir quase todos os problemas ou eventos inesperados, porém nesses casos, uma intervenção humana (como diagnóstico e solução em campo) provavelmente será necessária, o que pode ser indesejável pela localização dos módulos em plataforma flutuante.

## 6.4 Dados coletados

Infelizmente, devido à demora na chegada dos componentes eletrônicos para a montagem da placa de circuito impresso, ainda não foi possível sua implementação, o que também prejudicou a etapa posterior, que seria a etapa de ajustes finais do sistema. Por essas razões, o sistema não realizou coleta de dados em campo, sendo limitada apenas à sala em que estava guardado, mas em operação/desenvolvimento.

Apesar disso, alguns dados coletados podem ser utilizados em gráficos apenas para demonstrar que a coleta realmente faz algum sentido, o que é essencial para o estudo científico, pois conclusões serão feitas após aplicar cálculos estatísticos nesses dados. As Figuras 55 e 56 demonstram alguns gráficos gerados com alguns dos vários dados coletados pelo sistema ao mesmo tempo, que são do dia 1 de Junho de 2024.

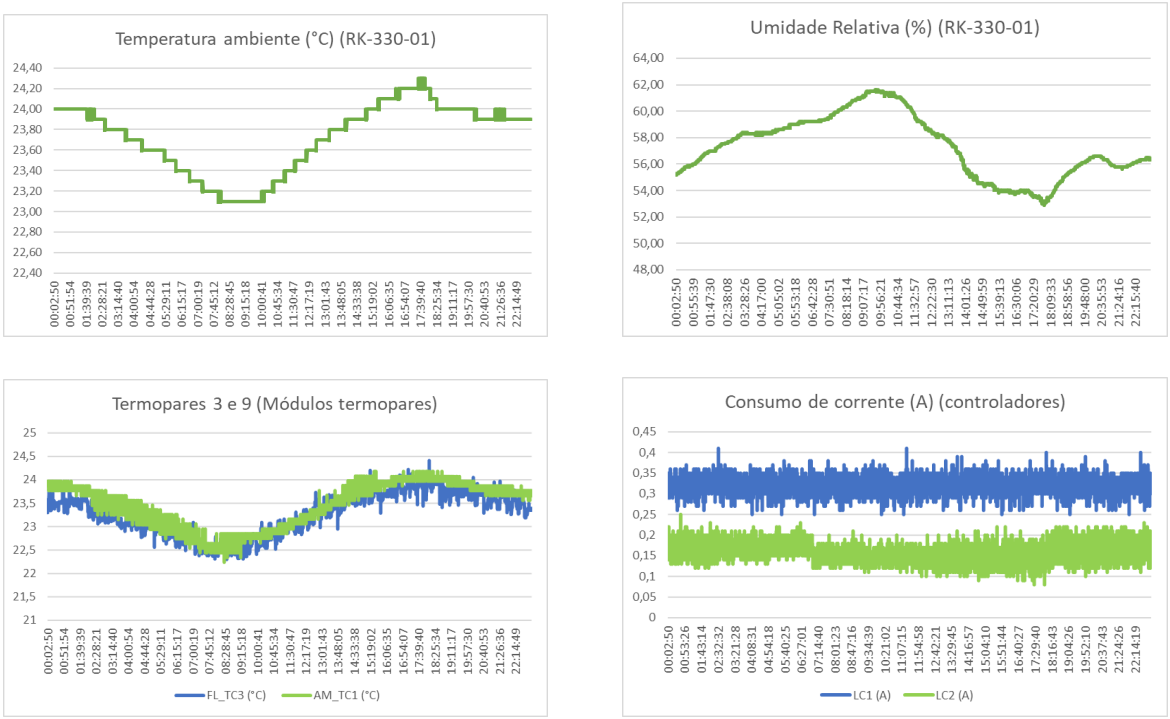



Figura 55 – Curvas obtidas de sensores 1

Fonte: Produzida pelo autor

A Figura 55 apresenta duas curvas que demonstram o comportamento climático da sala onde o sistema estava em operação, onde é possível observar um declínio da temperatura a partir de 0:02, até por volta das 8:28, onde a temperatura da sala é a menor (23 °C). O pico de temperatura acontece por volta das 17h, e a temperatura volta a estabilizar para em torno de 24 °C. É possível observar que a temperatura da sala não tem mudança significativa, variando muito pouco ao longo do dia. A umidade já é um pouco mais variável, tendo quase 62% como valor máximo por volta das 9h, e chegando a um valor mínimo por volta das 17h. Os valores de temperatura para os termopares também não apresentam variação significativa, apesar do ruído na medição ser perceptível. Na Figura 55, os termopares comparados são da face superior do módulo superior (curva verde, termopar 3) e da face inferior do módulo inferior (curva azul, termpar 9). O efeito de degrau deve-se ao período de amostragem, que é de aproximadamente 10 segundos. Se não há variação da grandeza, uma variação brusca seguida por ausência de variação causará esse efeito. É possível ver que nem todas as curvas apresentam esse efeito.

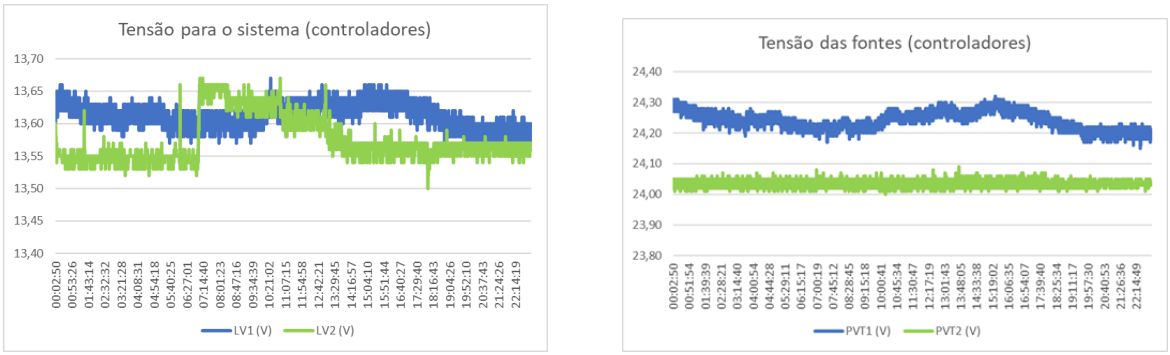



Figura 56 – Curvas obtidas de sensores 2

Fonte: Produzida pelo autor

Nas duas últimas figuras da Figura 55 e na Figura 56, é possível observar que o consumo de corrente do controlador 1 (curva azul) está em cerca de 350 mA, com um valor máximo de 400 mA, aproximadamente. Esse consumo de corrente é em grande parte devido ao *Raspberry Pi*. Por outro lado, o quadro que possui o controlador 2 (curva verde), responsável pela alimentação do conjunto de sensores apresenta um consumo menor, com um valor máximo de 200 mA, aproximadamente. Das duas últimas curvas observadas na Figura 56, só é possível concluir que a fonte de alimentação (simulando um módulo fotovoltaico) ligada ao controlador 1 (curva azul PVT1) oscila mais, o que de fato acontece, por observação em laboratório. A fonte ligada ao controlador 2 não oscila tanto.

Da outra grande quantidade de dados coletados que não estão demonstrados aqui, não há análises úteis a se fazer. A maioria dos dados coletados quando os módulos não estão em operação levam conclusões triviais, porém coerentes. Uma análise em campo deverá trazer conclusões e correlações mais interessantes. Além disso, uma quantidade desses dados, mesmo sendo lidas de componentes que apresentam leituras erradas, serviram para validação do sistema. Por fim, pode-se afirmar que o sistema já apresenta o funcionamento desejado no que se diz respeito à integração, que é o maior objetivo desse trabalho.

# 7 Conclusões

Pode-se considerar que a grande maioria dos objetivos mencionados na Introdução desse trabalho foram atingidos. O maior objetivo era, a princípio, o desenvolvimento de um *software* que fosse capaz de integrar a comunicação com todos os barramentos instalados no sistema, e reunir uma grande quantidade de dados provindos de diversos sensores em um lugar de fácil acesso pelo usuário, onde poderiam ser processados posteriormente. A seguir, era necessário que esse *software* fosse capaz de tornar a intervenção humana quase que desnecessária no sistema como um todo, enviando essa grande quantidade de dados periodicamente à nuvem, restando ao usuário final o trabalho simples de processar esses dados para tirar conclusões úteis a respeito do desempenho das microusinas fotovoltaicas *off-grid*. Foi visto anteriormente que isso de fato acontece.

A camada de gerenciamento adicional implementada, o *design* da implementação e a placa integradora são adicionais que não deixam de ter sua importância. A camada de gerenciamento, por garantir o funcionamento correto do sistema, a placa integradora e o projeto do *software*, para fins de conveniência.

Observa-se que a grande maioria dos objetivos foram concretizados. A próxima etapa seria justamente a instalação do sistema no ambiente proposto e a verificação dos dados coletados, etapas indispensáveis para a validação do projeto. Essas etapas finais, ao serem concluídas, representam a consolidação do projeto de engenharia demonstrado aqui. Após verificado funcionamento correto, os dados fornecidos coletados dos módulos serão de grande utilidade para estudos científicos, o que demonstra a importância do projeto apresentado aqui.

# 8 Trabalhos futuros

Após a integração realizada, resta a instalação e validação do sistema em ambientes fixo e em corpo d'água. É a etapa mais importante desse projeto e consiste em sua consolidação. Poderá ser realizado após instalação da placa integradora e correções mínimas no sistema no que se refere à instrumentação e configurações internas ao *software* de automação.

O *software* de automação pode sofrer alterações mínimas devido à necessidade de adaptações conforme mudanças na instrumentação forem realizadas, pois cada sistema tem seu conjunto individual de sensores e suas variáveis de condição, que serão ativadas ou não conforme o sistema. Além disso, otimizações no código também podem ser realizadas, cabendo ao desenvolvedor implementá-las conforme o necessário. Atualmente, não há essa necessidade.

Outro trabalho interessante é o desenvolvimento de uma interface cliente-servidor que possibilite o acesso remoto do sistema via *browser*. Isso requer conhecimentos de rede e desenvolvimento *web*, que não estão muito relacionados à instrumentação, mas cuja interface seria útil, pois descartaria a necessidade da utilização de *software* de terceiros como o *TeamViewer*.

Por mais que a memória não seja um problema nesse escopo, a implementação de um banco de dados tradicional é uma ideia interessante, pois a quantidade de dados coletada pelo sistema é muito grande e necessita organização e tratamento para posterior processamento.

Por fim, a implementação de uma malha de controle de vazão e resistência de consumo de excedente de geração com o microcomputador *Raspberry Pi* resultará em um consumo de energia mais apropriado, isto é, toda energia gerada poderá ser consumida com a variação da corrente consumida pela resistência extra de dissipação de excedente de geração. Quanto à vazão, tem a ver com a quantidade de fluido refrigerante entrando no sistema trocador de calor por segundo, que sendo variada poderia tornar o estudo científico um pouco mais interessante, já que a refrigeração dos módulos depende dessa grandeza.

# Referências


ADVANIO TECH CO., LTD. **W-M1B103 3 in 1 Universal Analog Input Module Datasheet**. Citado na p. 37.

ALMEIDA, V. G. R. d. **Integração de instrumentação e capacidade de comunicação com nuvem a uma placa Raspberry Pi para monitoramento remoto de plantas de geração fotovoltaicas off-grid**. 2022. Universidade de Brasília. Disponível em: <https://bdm.unb.br/handle/10483/34440>. Citado nas pp. 17, 27, 37.

ANALOG DEVICES. **3 V/5 V, 1 mW, 2-/3-Channel, 16-Bit, Sigma-Delta ADCs - AD7705-AD7706**. 2006. Disponível em: <https://www.analog.com/media/en/technical-documentation/data-sheets/ad7705_7706.pdf>. Citado nas pp. 54, 55.

BARBOSA, G. V.; ALCÂNTARA, M. F. S.; ARAÚJO, M. R. G.; FONSECA, T. T.; RODRIGUES, R. F. N.; DIAS, M. J. Análise de Eficiência Solar em Painéis Fotovoltaicos. **Revista Processos Químicos**, v. 14, n. 28, p. 108–115, abr. 2021. DOI: 10.19142/rpq.v14i28.607. Disponível em: <http://ojs.rpqsenai.org.br/index.php/rpq_n1/article/view/607>. Citado na p. 16.

BERG, J. **MinimalModbus documentation**. 2023. Disponível em: <https://minimalmodbus.readthedocs.io/en/stable/>. Citado na p. 51.

BRONZATTI, F. L.; IAROZINSKI NETO, A. Matrizes energéticas no Brasil: cenário 2010-2030. **Encontro Nacional de Engenharia de Produção**, v. 28, p. 13–16, 2008. Disponível em: <https://www.fans.edu.br/wp-content/uploads/2015/06/texto_matrizes_energeticas_brasil_cenario_2010.2030.pdf>. Citado na p. 15.

CYPRESS SEMICONDUCTOR CORPORATION. **OneWire Datasheet - OneWire V 1.1**. 2014. Disponível em: <https://www.infineon.com/dgdl/Infineon-OneWire_001-43362-Software%20Module%20Datasheets-v01_01-EN.pdf?fileId=8ac78c8c7d0d8da4017d0f987bd907a0>. Citado na p. 20.

DAWOUD SHENOUDA DAWOUD, P. D. **Serial Communication Protocols and Standards (River Publishers Series in Communications)**. River Publishers, 2020. ISBN 9788770221542. Citado nas pp. 19, 22, 23.

FRENZEL, L. **Handbook of Serial Communications Interfaces: A Comprehensive Compendium of Serial Digital Input/Output I/O) Standards**. 1. ed.: Newnes, 2015. ISBN 9780128006290. Citado nas pp. 20, 21.

HUNAN RIKA ELECTRONIC TECHNOLOGY CO., LTD. **RK120-01C Combined Wind Speed & Direction Sensor**. 2015a. Version 3.0. Citado na p. 29.

HUNAN RIKA ELECTRONIC TECHNOLOGY CO., LTD. **RK200-03 Pyranometer(class one)**. 2015b. Version 3.0. Citado na p. 31.

HUNAN RIKA ELECTRONIC TECHNOLOGY CO., LTD. **RK200-04 Solar Radiation Sensor**. 2015c. Version 3.0. Citado na p. 32.

HUNAN RIKA ELECTRONIC TECHNOLOGY CO., LTD. **RK300-02 Dust Concentration Sensor**. 2015d. Version 3.0. Citado na p. 28.

HUNAN RIKA ELECTRONIC TECHNOLOGY CO., LTD. **RK330-01 Atmospheric Temperature, Humidity & Pressure Sensor**. 2015e. Version 3.0. Citado na p. 30.

HUNAN RIKA ELECTRONIC TECHNOLOGY CO., LTD. **RK900-09 Miniature Ultrasonic Automatic Weather Instrument**. 2015f. Version 3.0. Citado na p. 33.

MACIEL, A. J. L. **Desenvolvimento de instrumentação para monitoramento de plantas de geração fotovoltaica off-grid, instaladas em solo e sobre futuadores em corpo d'água**. 2021. Universidade de Brasília. Disponível em: <https://bdm.unb.br/handle/10483/32463>. Citado nas pp. 17, 27.

MANKAR, J.; DARODE, C.; TRIVEDI, K.; KANOJE, M.; SHAHARE, P. Review of I2C protocol. **International Journal of Research in Advent Technology**, Citeseer, v. 2, n. 1, 2014. Disponível em: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=314537daa1f601f83044b25b68e2af6c8f331f3f>. Citado na p. 21.

PEREIRA, E. B. ; M.; GONÇALVES, F. R. ; COSTA, A. R. ; LIMA, R. S. ; RÜTHER, F. L. ; ABREU, R. ; TIEPOLO, S. L. ; PEREIRA, G. M. ; SOUZA, S. V. ; G., J. **Atlas brasileiro de energia solar 2. ed**. São José dos Campos: INPE, 2017. Disponível em: <http://doi.org/10.34024/978851700089>. Citado na p. 15.

RASPBERRY PI LTD. **Raspberry Pi Sense HAT - Product Brief**. 2023. Citado na p. 39.

REZENDE, J. O. A importância da Energia Solar para o Desenvolvimento Sustentável. **São Paulo: Atena**, 2019. Disponível em: <https://www.atenaeditora.com.br/catalogo/download-file/2584>.

SHINGARE, T. D.; PATIL, R. SPI implementation on FPGA. **International Journal of Innovative Technology and Exploring Engineering (IJITEE)**, Citeseer, v. 2, n. 2, p. 7–9, 2013. Disponível em: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=660e2932d0ddbf2a188c442ad81e4551c6bd7ff9>. Citado na p. 22.

TECNOLOG LTDA. **AM8T - Módulo de termopares - Manual do Usuário**. 2021. Citado na p. 36.

THE MODBUS ORGANIZATION. **Modbus Application Protocol Specification v1.1b3**. Abr. 2012. Disponível em: <https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf>. Citado nas pp. 24–26.

THE MODBUS ORGANIZATION. **MODBUS over Serial Line Specification and Implementation Guide V1.02**. Dez. 2006. Disponível em: <https://modbus.org/docs/Modbus_over_serial_line_V1_02.pdf>. Citado nas pp. 19, 23.

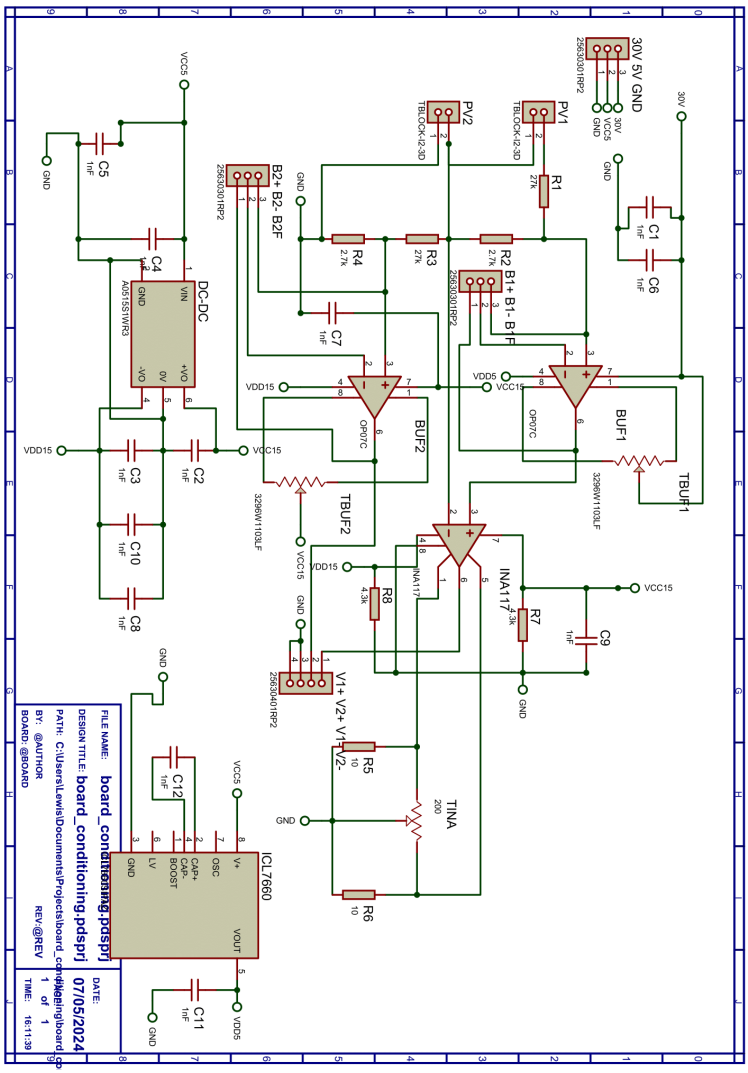
ZILLES, R. Energia solar fotovoltaica. **USP, São Paulo**, 2012. Disponível em: <https://www.iee.usp.br/sites/default/files/biblioteca/producao/2012/Livros/zillesenergiasolar.pdf>.

# Apêndices

# Apêndice A – Diagramas e esquemáticos

## A.1 Esquemático do projeto da placa integradora

Acesse o projeto da placa pelo link https://drive.google.com/file/d/13pNwAtCW_9FjCPFQCOFnPPJ-ZdqW3ydr/view?usp=sharing. É necessário pedir permissão para acesso.



# Apêndice B – Códigos de programação

## B.1 Códigos de exemplo

Código 5 – Código de exemplo para leitura de tensões e correntes da associação de painéis

```
import pigpio
import os
import time
from AD770X import *

#Carrega biblioteca de GPIO e abstrai o AD7705 para um objeto
os.system('sudo pigpiod')
time.sleep(.5)
ad7705 = AD770X(device=0)
pi = pigpio.pi()

#Parametros eletricos do circuito
GPIO_TENSAO_P1P2 = 5
GPIO_CURR = 6
G_P25_P1 = (3.240/67.70)
G_P25_P2 = (4.639/46.74)
G_AD620_P1 = 1.45
OFF_AD620_P1 = 0.0235

#Pre-operacoes (otimizacao)
op_tp1 = ESCALA*(1/G_P25_P1)
op_tp2 = ESCALA*G_AD620_P1*(1/G_P25_P2)

def panels_read():
    while True:
        pi.write(GPIO_TENSAO_P1P2, 0) #Escolhe o AD7705 das tensoes utilizando o pino
            GPIO em BAIXO
        ad7705.initChannel(CHN_AIN1) #Escolhe qual o canal de leitura do AD7705
        #Le, processa e imprime os valores
        print('Tensao P1:',
            ((ad7705.readADResultRaw(CHN_AIN1)*ESCALA)-OFF_AD620_P1)*op_tp1)
        ad7705.initChannel(CHN_AIN2)
        print('Tensao P2:', ad7705.readADResultRaw(CHN_AIN2)*op_tp2)
        pi.write(GPIO_TENSAO_P1P2, 1) #Coloca o pino GPIO em ALTO para selecionar outro
            dispositivo
        #Analogo para a corrente
        pi.write(GPIO_CURR, 0) #Escolhe o AD7705 da corrente colocando o pino GPIO em
            BAIXO
        ad7705.initChannel(CHN_AIN1)
        print('Curr P1_P2:', ((ad7705.readADResultRaw(CHN_AIN1)*ESCALA)-2.5)*40/8)
        pi.write(GPIO_CURR, 1) #Coloca o pino GPIO em ALTO para selecionar outro
            dispositivo

panels_read()
```

Código 6 – Código de exemplo para leitura de grandezas com o SenseHat

```
from sense_hat import SenseHat
sense = SenseHat() #Instancia o modulo SenseHat com um objeto
sense.set_imu_config(True, True, True) #Escolhe quais sensores serao lidos (todos)

def sense_read():
    #Le os valores de todos os sensores (ambientais, giroscopio, acelerometro,
        magnetometro)
    umidade = '{:.2f}'.format(float(sense.get_humidity()))
```

```
    pressao = '{:.2f}'.format(float(sense.get_pressure()))
    compass = '{:.2f}'.format(float(sense.get_compass()))
    gyro = sense.get_gyroscope()
    gyro_roll = '{:.2f}'.format(float(gyro['roll']))
    gyro_pitch = '{:.2f}'.format(float(gyro['pitch']))
    gyro_yaw = '{:.2f}'.format(float(gyro['yaw']))
    accelerom = sense.get_accelerometer()
    accelerom_roll =  '{:.2f}'.format(float(accelerom['roll']))
    accelerom_pitch = '{:.2f}'.format(float(accelerom['pitch']))
    accelerom_yaw = '{:.2f}'.format(float(accelerom['yaw']))
    data = {"umidade_relativa(percent)": umidade, "pressao(milliBar)":
        pressao,"compass_norte(deg)": compass,"gyro_roll(deg)":
        gyro_roll,"gyro_pitch(deg)": gyro_pitch,"gyro_yaw(deg)":
        gyro_yaw,"accelerometer_roll(deg)": accelerom_roll,"accelerometer_pitch(deg)":
        accelerom_pitch,"accelerometer_yaw(deg)": accelerom_yaw}
    return data

#Imprime a leitura de valores continuamente
def example():
    try:
        while True:
            data = sense_read()
            for key, value in data.items():
                print(key+':', value),
            print('\n\n\n')
    except KeyboardInterrupt:
        print('End')

example()
```

Código 7 – Código de exemplo para comunicação modbus

```
import serial
import minimalmodbus
import time

port = '/dev/ttyUSB0' #Escolhe a porta onde esta conectado o barramento
devices_addr = {'RK120-01':6, 'Controlador':12} #Endereco do disp na rede
devices_reg = {'RK120-01':0, 'Controlador':12561} #Endereco do registrador inicial
devices_baud = {'RK120-01':19200, 'Controlador':115200} #Baudrate

def modbus_read(device, function, registers):
    instrument = minimalmodbus.Instrument(port, devices_addr[device], debug = True)
    instrument.serial.baudrate = devices_baud[device] #Configura os parametros da
        comunicacao serial
    instrument.serial.bytesize = 8
    instrument.serial.parity   = serial.PARITY_NONE
    instrument.serial.stopbits = 1
    instrument.serial.timeout  = .200
    instrument.mode = minimalmodbus.MODE_RTU
    instrument.clear_buffers_before_each_transaction = True
    instrument.serial.port
    instrument.address
    try:
        response = instrument.read_registers(devices_reg[device], registers, function)
        return True, response
    except:
        return False, 'NaN'

def device_read(device, function, registers):
    success, response = modbus_read(device, function, registers)
    while success == False:
        success, response = modbus_read(device, function, registers)
    print('Resposta do dispositivo:', response)

device_read('RK120-01', 3, 2)
device_read('Controlador', 4, 1)
```

Código 8 – Código de exemplo para comunicação em baixo nível utilizando o barramento SPI

```
import spidev
```

```
#Escolhe o barramento do dispositivo e configura os parametros da comunicacao SPI
bus = 0
device = 0
SPEED = 50000
BITS = 8
MODE = 0b11 #SPI_CPHA | SPI_CPOL

spi = spidev.SpiDev() #Instancia a comunicação SPI com o uso de uma classe
spi.open(bus, device) #Abre a comunicacao

#Aplica os parametros configurados
spi.max_speed_hz = SPEED
spi.mode = 0b11
spi.bits_per_word = BITS

#Envia ao dispositivo o valor hexadecimal 0x11
r = 0x11
spi.xfer([r])
```

Código 9 – Fragmento de código - Rotina de upload do software

```
def data_upload():
    print('Upload routine started, please wait...')
    cwd = os.getcwd()
    time_now = date.now()
    time_now = time_now.strftime('%d/%m/%y')
    time_now = time_now.replace('/','-')

    datapath = cwd+data_dir
    lowfreq_file = datapath+lowfreq_filename+time_now+".csv"
    highfreq_dir = datapath+highfreq_filename+time_now+r'/'
    data_folder = datapath+r'dados'+time_now
    os.system(f"mkdir {data_folder}")
    os.system(f"mv {lowfreq_file} {highfreq_dir} {data_folder}")
    #os.system(f"mv {lowfreq_file} {data_folder}")

    zipfile = "dados"+time_now+".zip"

    gauth=GoogleAuth()
    gauth.LocalWebserverAuth()  # Cria um servidor da web local e gerencia a
        autenticação automática
    drive = GoogleDrive(gauth)
    drive_file = drive.CreateFile({'parents':
        [{'id':'1A3uP92xXJKLdMz4-Mhz0APd5W9QFXPSK'}],'title': zipfile}) #Localiza pasta
        no GDrive

    zipfile = "dados"+time_now #Comprime arquivo
    shutil.make_archive(datapath+zipfile, 'zip', data_folder)
    zipfile = zipfile+".zip"
    drive_file.SetContentFile(datapath+zipfile) #Cria arquivo para upload
    drive_file.Upload() #Faz o upload
    print('Uploaded!')
    os.system(f"rm -r {data_folder}") #Remove arquivo do sistema
    return True #Não houveram erros
```

## B.2 Códigos do software de automação

O software de automação completo para instalação e em pleno funcionamento (desde que a instrumentação esteja presente) pode ser encontrado em: https://drive.google.com/file/d/1JgsqhvPgUvRFzs8JYYVpnwTvWGHoalL2/view?usp=sharing. É necessário pedir permissão para acessar e fazer *download*.

Código 10 – Código do módulo principal

```
"""
Author: Luiz Felipe Almeida Silva - 180023098

Main Module: main.py
Data Transfer Module: datapath.py
Sensors, Hardware and Communications Module: hardware.py
SenseHat and SPI Module: shspi.py
Watchdog Module: watchdog.py
GUI Module: gui.py

File: main.py

Description: This file controls what happens inside the program, in which order. It also
    contains
      some miscellaneous functions.

Date: 27/08/2023
"""
#Libraries
import os
import sys
import time
import signal
import threading as thr
import multiprocessing as mp
import hardware as hardw
import datapath as data
import shspi as ss
import watchdog as wd
import gui
from datetime import datetime as date

#Global
#Timers
lf_timer = 0.0
lf_start = 0.0
lf_end = 0.0

#Hardware
variables = []
ow_devices = []
ow_variables = []
modbus_devices = []

#Processes
proc_wd = []
proc_sh = []
proc_spi = []
proc_main = []
q_watchdog = mp.JoinableQueue() #Watchdog queue, number 1
q_main = mp.JoinableQueue() #Main queue, number 2
q_except = mp.JoinableQueue() #Exception queue, number 3
q_spi = mp.JoinableQueue() #SPI queue, number 4
q_sh = mp.JoinableQueue() #SenseHat queue, number 5

#Returns hour
def get_time():
  hour_str = date.now()
  hour_str = hour_str.strftime('%H:%M:%S')
  return hour_str

#Loads information about the sensors
def system_initialize():
  global modbus_devices
  global ow_devices
  global ow_variables
  global variables

  aux = []
  aux.append('Hora')
  aux.append('Elapsed(s)')
  modbus_devices = hardw.devices_load()
  for i in modbus_devices:
```

```
    variables, error = hardw.variables_parse(i)
    aux += variables

  ow_devices, ow_variables = hardw.owire_load()
  variables = aux
  if hardw.is_earth == False:
    variables += ow_variables
  #print(modbus_devices)
  modbus_devices = hardw.sensors_test(modbus_devices, ow_devices, ow_variables)
  #print(modbus_devices)
  variables = []
  aux = []
  aux.append('Hora')
  aux.append('Elapsed(s)')

  for i in modbus_devices:
    variables, error = hardw.variables_parse(i)
    aux += variables

  ow_devices, ow_variables = hardw.owire_load()
  variables = aux
  if hardw.is_earth == False:
    variables += ow_variables

#Lowfreq loop
def lowfreq_read():
  global q_watchdog
  global q_except

  try:
    time.sleep(2)
    gui.send_console('Please wait while the information is gathered...\n', gui.console)
    os.system('clear')
    faulty = []
    while True:
      lf_start = time.time()
      aux = {} #Information list
      info, faulty = hardw.master_routine(modbus_devices, ow_devices, ow_variables) #Get
          info from devices using the master
      hour_str = get_time()
      aux.update({'Hora':hour_str})
      aux.update(info) #Append information
      gui.send_console('Information gathered. Waiting to post...\n', gui.console)
      q_watchdog.put(aux)
      q_watchdog.put(faulty)
      q_watchdog.join()
      time.sleep(5) # Wait to take data again: sampling interval around 10 s
      #time.sleep(9) # Wait to take data again: sampling interval around 15 s
      #time.sleep(14) # Wait to take data again: sampling interval around 20 s
      lf_end = time.time()
      aux.update({'Elapsed(s)':round(lf_end-lf_start, 2)})
      data.sensors_writedata(variables, aux) #Writes the data to the csv file.
      gui.post_information(variables, aux, gui.labels)
      if q_except.empty() == False:
        command = q_except.get()
        if command == 0:
          system_initialize()
          faulty = []
  except:
    print('Low frequency loop ended due to exception or killing.')
    data.log_write('Low frequency loop ended due to exception or killing.')
    return

#Main threads
def mainproc(q2):
  pid_wd = q2.get()
  pid_spi = q2.get()
  thr_lowfreq = thr.Thread(target = lowfreq_read)
  gui.initialize(gui.w_def, variables)
  thr_lowfreq.start()
  gui.start()
  #If windows closes, all processes get killed.
  os.kill(pid_wd, signal.SIGTERM)
  os.kill(pid_spi, signal.SIGTERM)
```

```
  if not hardw.is_earth:
    pid_sh = q2.get()
    os.kill(pid_sh, signal.SIGTERM)
  os.kill(os.getpid(), signal.SIGTERM) #Kills itself

#Close system utilities
def system_cleanup():
  print('Cleaning up system.')
  os.system('sudo killall pigpiod')
  hardw.GPIO.cleanup()
  data.log_write('System was cleaned up.')

#Setup and shutdown rotines
def main():
  global proc_main
  global proc_spi
  global proc_sh
  global proc_wd

  #Sys settings
  if len(sys.argv) > 1 and len(sys.argv) < 3:
    if sys.argv[1] == 'fixo':
      print('System: Fixed')
      hardw.is_earth = True
    elif sys.argv[1] == 'flut':
      print('System: Floating')
      hardw.is_earth = False
      hardw.filename_settings = "modbus_flut"+".csv"
    else:
      print('Invalid arguments. Usage: main.py (fixo | flut)')
      return
  else:
    print('Invalid usage. Usage: main.py (fixo | flut)')
    return

  system_initialize()
  #system_cleanup()
  #return
  hardw.initialize(q_watchdog)
  ss.initialize(hardw.is_earth)
  data.initialize()

  proc_main = mp.Process(target=mainproc, args=(q_main,))
  proc_wd = mp.Process(target = wd.main, args=(q_watchdog, q_except, q_spi, q_sh,))

  proc_spi = mp.Process(target=ss.spi_read, args=(hardw.is_earth, q_spi))
  if hardw.is_earth == False:
    proc_sh = mp.Process(target=ss.sh_read, args=(hardw.is_earth, q_sh))
    proc_sh.start()
  proc_spi.start()
  #Those above are separated due to faster rates.

  proc_main.start()
  q_watchdog.put(modbus_devices)
  q_watchdog.put(ow_devices)
  q_watchdog.put(variables)
  q_watchdog.put(proc_main.pid)
  q_watchdog.put(proc_spi.pid)
  if not hardw.is_earth:
    q_watchdog.put(proc_sh.pid)
    q_main.put(proc_sh.pid)
  proc_wd.start()
  q_main.put(proc_wd.pid)
  q_main.put(proc_spi.pid)
  data.log_write('System started successfully.')
  proc_main.join()
  proc_wd.join()
  system_cleanup()

if __name__ == "__main__":
  main()
```

Código 11 – Código do módulo de hardware

```
"""
File: hardware.py

Description: This file is responsible for facilitating communication
             between the electronic instrumentation used in the project.

"""
#Libraries
import os
import csv
import minimalmodbus
import serial
import time
import numpy as np
import gui
import psutil
import datapath as data
import RPi.GPIO as GPIO
#from collections import OrderedDict

is_earth = False

#File settings
filename_settings = "modbus_fixo"+".csv" #Sensor file
header_fields = ['nome','MODBUS_ADDRESS','Q_registradores','frequencia','funcao_leitura',
         'registrador_inicio','enderecos_reg','fator_mutiplicador','names_reg']

#Relays
bomba1 = 11 #Relay 1 - backsheet pump NO
bomba2 = 13 #Relay 2 - curtain pump NO
bomba3 = 36 #Relay 3 - heat exchanger NO
sinalizador = 8 # Relay 4 signal lights NO
reset_modbus = 10 # Relay 5 sensor hard reset NC
vent_1 = 12 # Relay 6 fan 1 - rasp NO
vent_2 = 26 # Relay 7 fan 2 - controller NO

#Queues
q_watchdog = 0

#GPIO setup
GPIO.setmode(GPIO.BOARD)
GPIO.setup(bomba1, GPIO.OUT)
GPIO.setup(bomba2, GPIO.OUT)
GPIO.setup(bomba3, GPIO.OUT)
GPIO.setup(sinalizador, GPIO.OUT)
GPIO.setup(reset_modbus, GPIO.OUT)
GPIO.setup(vent_1, GPIO.OUT)
GPIO.setup(vent_2, GPIO.OUT)

def initialize(q1):
  global q_watchdog
  q_watchdog = q1
  #LOW = Coil energized
  GPIO.output(bomba1, GPIO.HIGH)
  GPIO.output(bomba2, GPIO.HIGH)
  GPIO.output(bomba3, GPIO.HIGH)
  GPIO.output(sinalizador, GPIO.HIGH)
  GPIO.output(reset_modbus, GPIO.HIGH) #Modbus on (NC)
  GPIO.output(vent_1, GPIO.HIGH)
  GPIO.output(vent_2, GPIO.HIGH)

def modbus_hard_reset():
  GPIO.output(reset_modbus, GPIO.LOW)
  time.sleep(5)
  GPIO.output(reset_modbus, GPIO.HIGH)

def get_cpu_temp():
    temp = psutil.sensors_temperatures()['cpu_thermal'][0].current
    return temp

#Parses Modbus information
def modbus_save(updated_info, info_list, success, factor):
```

```
  info = {}
  if success: #If there was no errors
    for i in range(0, len(updated_info)):
      aux = round(np.float32(np.int16(updated_info[i]))*np.float32(factor[i]), 3)
      aux = {str(info_list[i]):aux}
      info.update(aux)
    return info
  else: #If the connection failed
    for i in range(0, len(info_list)):
      aux = {str(info_list[i]):'NaN'} #Change to NaN
      info.update(aux)
    return info

#Parse the register variables parameters from a vector.
def variables_parse(fields):
  name = eval(fields['names_reg'])
  factor = eval(fields['fator_mutiplicador'])
  return name, factor

#Read devices registers
def modbus_read(device, reg, factor):
  port = '/dev/ttyUSB0'
  try:
    #If debug, it will print this
    instrument = minimalmodbus.Instrument(port, int(device['MODBUS_ADDRESS']), debug =
        False)
    instrument.serial.baudrate = int(device['frequencia'])       # Baud
    instrument.serial.bytesize = 8
    instrument.serial.parity   = serial.PARITY_NONE
    instrument.serial.stopbits = 1
    instrument.serial.timeout  = .200  # seconds
    instrument.mode = minimalmodbus.MODE_RTU   # rtu or ascii mode
    instrument.clear_buffers_before_each_transaction = True
    instrument.serial.port                     # this is the serial port name
    instrument.address                         # this is the slave address number

    if(int(device['funcao_leitura']) == 3):
      aux = instrument.read_registers(int(device['registrador_inicio']),
                  int(device['Q_registradores']), int(device['funcao_leitura']))
      aux = modbus_save(aux, reg, True, factor)
    elif(int(device['funcao_leitura']) == 4):
      aux2 = []
      for i in eval(device['enderecos_reg']):
        temp = instrument.read_registers(int(i), 1, 4)
        temp = temp.pop(0)
        aux2.append(temp)
        aux = modbus_save(aux2, reg, True, factor)
    return aux
  except:
    aux = []
    aux = modbus_save(aux, reg, False, factor)
    return False

#Read devices information from a file
def devices_read(cabecario,name):
        aux = []
        cwd = os.getcwd()
        fileName=cwd+'/sensors_config/'+name
        if os.path.exists(fileName):
            with open(fileName) as csvfile:
                csv_reader = csv.DictReader(csvfile, fieldnames=cabecario)
                csv_reader.__next__()
                for row in csv_reader:
                    aux.append(row)
        return aux, os.path.exists(fileName)

#Load device names with regs
def devices_load():
  aux, error = devices_read(header_fields, filename_settings)
  devices = [] #device list
  for i in aux:
    devices.append(i)
  return devices
```

```
def owire_load():
  devices_id = []
  devices_var = []
  aux, error = devices_read(['name','id'], 'owire.csv')
  aux = list(aux)
  for i in aux:
    devices_id.append(i['id'])
    devices_var.append(i['name'])
  return devices_id, devices_var

def owire_watersensor(id):
  try:
    aux = ''
    filename = 'w1_slave'
    f = open('/sys/bus/w1/devices/' + id + '/' + filename, 'r')
    line = f.readline() # read 1st line
    crc = line.rsplit(' ',1)
    crc = crc[1].replace('\n', '')
    if crc=='YES':
      line = f.readline() # read 2nd line
      aux = line.rsplit('t=',1)
    else:
      f.close()
      return 'NaN'
    f.close()
    return float(aux[1])
  except:
    return 'NaN'

def owire_data(names, id):
  T = []
  for i in id:
    aux = owire_watersensor(i)
    if aux != 'NaN':
      aux = aux/1000  #Water sensor conversion factor
      aux = '{:.2f}'.format(float(aux))
    T.append(aux)

  info = {}
  for i in range(0, 4):
    aux = {names[i]:T[i]}
    info.update(aux)
  return info

def sensors_test(modbus_sensors, ow_sensors, ow_variables):
  faulty = set()
  is_faulty = data.json_read('sensors_faulty.json')
  for i in modbus_sensors:
    if bool(is_faulty[i['nome']]):
      reg, mult = variables_parse(i)
      temp = modbus_read(i, reg, mult)
      counter = 0
      while temp == False:
        temp = modbus_read(i, reg, mult)
        counter += 1
        if counter >= 5:
          msg = f"Sensor {i['nome']} address {i['MODBUS_ADDRESS']} is not responding and
              thus will not be read."
          faulty.add(i['nome'])
          data.log_write(msg)
          break
      counter = 0
    else:
      continue

  aux = []
  for item in modbus_sensors:
    if item.get('nome') not in faulty:
      aux.append(item)
  return aux


def master_routine(modbus_devices, ow_devices, ow_variables):
  aux = {}
```

```
  faulty = []
  for i in modbus_devices:
    reg, mult = variables_parse(i)
    temp = modbus_read(i, reg, mult)
    counter = 0
    while temp == False:
      temp = modbus_read(i, reg, mult)
      counter += 1
      if counter >= 5:
        msg = f"Sensor {i['MODBUS_ADDRESS']} {i['nome']} did not respond after {counter}
            times. Ignoring\n"
        #print('Sensor', i['MODBUS_ADDRESS'], i['nome'], 'did not respond after',
            counter, 'times. Ignoring')
        gui.send_console(msg, gui.console)
        faulty.append(i['nome'])
        temp = modbus_save(aux, reg, False, i['fator_mutiplicador'])
        break
    aux.update(temp)
    counter = 0
  if is_earth == False:
    temp = owire_data(ow_variables, ow_devices)
  aux.update(temp)
  gui.clear_console(gui.console)
  return aux, faulty
```

Código 12 – Código do módulo SPI/SenseHat

```
"""
File: shspi.py

Description: This file is responsible for SPI and SenseHat data acquisition system.

"""
#Libraries
import os
import time
import datetime
import pigpio
import datapath
from AD770X import *

#Misc variables
dir_folder = "/home/pi/Proj_FAPDF_PVF/dados/" #Change if program not at default dir
    (read the manual)

# pigpio lib and AD7705 start
os.system('sudo pigpiod')
time.sleep(.5)
ad7705 = AD770X(device=0)
pi = pigpio.pi()

#SPI Params definition
G_P25_P1 = 0
G_P25_P2 = 0
G_P25_P3 = 0
G_P25_P4 = 0
G_AD620_P1 = 0
G_AD620_P3 = 0
G_AD620_IRR = 0
G_AD260_IRR = 0
OFF_AD620_P1 = 0
OFF_AD620_P3 = 0
OFF_AD620_IRR = 0
GPIO_TENSAO_P1P2 = 0
GPIO_TENSAO_P3P4 = 0
GPIO_CURR = 0
GPIO_IRR = 0
GPIO_TENSAO = 0
LER_IRR = 0

#SenseHat
sense = 0
```

```
#header_sense = ["horario", "umidade_relativa(percent)", "temperatura_umidade(deg_C)",
    "temperatura_pressao(deg_C)", "pressao(milliBar)", "compass_norte(deg)",
    "gyro_roll(deg)", "gyro_pitch(deg)", "gyro_yaw(deg)", "accelerometer_roll(deg)",
    "accelerometer_pith(deg)", "accelerometer_yaw(deg)"]
header_sense = ["horario", "elapsed(ms)", "umidade_relativa(percent)",
    "pressao(milliBar)", "compass_norte(deg)", "gyro_roll(deg)", "gyro_pitch(deg)",
    "gyro_yaw(deg)", "accelerometer_roll(deg)", "accelerometer_pith(deg)",
    "accelerometer_yaw(deg)"]
title_sense = 'SenseHat - Plataforma Flutuante, Barragem Água Limpa, FAL-UnB'

#SPI File Parameters
header = ["horario", "elapsed(ms)", "tensao_p1(V)", "tensao_p2(V)", "tensao_p3(V)",
    "tensao_p4(V)", "corrente_p1p2(A)", "corrente_p3p4(A)", "irradiacao(W/m2)"]
spi_title_flutuante = 'SPI - Plataforma Flutuante, Barragem Água Limpa, FAL-UnB'
spi_title_fixo = 'SPI - Plataforma Fixa sobre laje, Campus Universitário Darcy Ribeiro,
    FT-ENM, Bloco G'

# Escalas comuns
ACS712_ESCALA = 0
ACS712_OFFSET = 0
fator_correcao = 22.13

#SPI Pre-Operations
op_tp1 = 0
op_tp2 = 0
op_tp3 = 0
op_tp4 = 0
op_irr = 0

#Timers
spi_timer = 0.0
sh_timer = 0.0
spi_start = 0.0
spi_end = 0.0
sh_start = 0.0
sh_end = 0.0

#Queues
q_sh = []
q_spi = []

def initialize(is_earth):
    global header
    global sense

    #Fixed
    global GPIO_TENSAO
    global LER_IRR
    global G_AD260_IRR

    #Float
    global GPIO_TENSAO_P1P2
    global GPIO_TENSAO_P3P4
    global GPIO_CURR
    global GPIO_IRR #Also fixed
    global G_P25_P1 #Also fixed
    global G_P25_P2 #Also fixed
    global G_P25_P3
    global G_P25_P4
    global G_AD620_P1 #Also fixed
    global G_AD620_P3
    global G_AD620_IRR #Also fixed
    global OFF_AD620_P1 #Also fixed
    global OFF_AD620_P3
    global OFF_AD620_IRR #Also fixed

    #Other
    global ACS712_ESCALA
    global ACS712_OFFSET
    global op_tp1
    global op_tp2
    global op_tp3
    global op_tp4
```

```
# SPI Params for Floating
    if is_earth == False:
        from sense_hat import SenseHat
        sense = SenseHat()
        sense.set_imu_config(True, True, True)

        params = datapath.json_read('flut_params.json')

        G_P25_P1 = params['G_P25_P1']
        G_P25_P2 = params['G_P25_P2']
        G_P25_P3 = params['G_P25_P3']
        G_P25_P4 = params['G_P25_P4']
        G_AD620_P1 = params['G_AD620_P1']
        G_AD620_P3 = params['G_AD620_P3']
        G_AD620_IRR = params['G_AD620_IRR']
        OFF_AD620_P1 = params['OFF_AD620_P1']
        OFF_AD620_P3 = params['OFF_AD620_P3']
        OFF_AD620_IRR = params['OFF_AD620_IRR']
        GPIO_TENSAO_P1P2 = params['GPIO_TENSAO_P1P2']
        GPIO_TENSAO_P3P4 = params['GPIO_TENSAO_P3P4']
        GPIO_CURR = params['GPIO_CURR']
        GPIO_IRR = params['GPIO_IRR']
        ACS712_ESCALA = params['ACS712_ESCALA'] #1A/100mV = 10A/V
        ACS712_OFFSET = params['ACS712_OFFSET'] #2.5V para 0A com ganho de 100mV/

    else:
        header.clear()
        header = ["horario", "elapsed(ms)", "tensao_p1(V)", "tensao_p2(V)",
            "irradiacao(W/m2)"]

        params = datapath.json_read('fixo_params.json')

        G_P25_P1 = params['G_P25_P1']
        G_P25_P2 = params['G_P25_P2']
        G_AD620_P1 = params['G_AD620_P1']
        G_AD260_IRR = params['G_AD260_IRR']
        OFF_AD620_P1 = params['OFF_AD620_P1']
        OFF_AD620_IRR = params['OFF_AD620_IRR']
        GPIO_TENSAO = params['GPIO_TENSAO']
        GPIO_IRR = params['GPIO_IRR']
        LER_IRR = bool(params['LER_IRR'])
        ACS712_ESCALA = params['ACS712_ESCALA'] #1A/100mV = 10A/V
        ACS712_OFFSET = params['ACS712_OFFSET'] #2.5V para 0A com ganho de 100mV/

    op_tp1 = (1/G_AD620_P1)*(1/G_P25_P1)
    op_tp2 = ESCALA*(1/G_P25_P2)
    op_tp3 = (1/G_AD620_P3)*(1/G_P25_P3)
    op_tp4 = ESCALA*(1/G_P25_P4)
    op_irr = ESCALA*(1/G_AD620_IRR)


def get_daypart(horario):
    if int(horario.strftime('%H')) >=0 and int(horario.strftime('%H')) < 6:
        day_part='1'
    elif int(horario.strftime('%H')) >=6 and int(horario.strftime('%H')) < 12:
        day_part='2'
    elif int(horario.strftime('%H')) >=12 and int(horario.strftime('%H')) < 18:
        day_part='3'
    elif int(horario.strftime('%H')) >=18:
        day_part='4'
    return day_part

def read_senseHat_sensors():
    global sense
    global sh_start
    global sh_end
    data = datetime.date.today()
    horario = datetime.datetime.now()

    day_part = get_daypart(horario)

    horario = horario.strftime('%H:%M:%S')
    umidade = '{:.2f}'.format(float(sense.get_humidity()))
    pressao = '{:.2f}'.format(float(sense.get_pressure()))
```

```
    compass = float(sense.get_compass())
    '''
    #Norte geografico na localidade de brasilia em 2024 tem um offset de approx. 22
        graus!
    Ou seja, norte geografico = azimute 22, relativo ao norte magnetico
    compass = compass - fator_correcao  # So LED appears to follow North

    if compass < 0:
        compass = compass+360
    if compass > 360:
        compass = compass-360
    '''
    compass = '{:.2f}'.format(float(compass))
    gyro = sense.get_gyroscope()
    gyro_roll = '{:.2f}'.format(float(gyro['roll']))
    gyro_pitch = '{:.2f}'.format(float(gyro['pitch']))
    gyro_yaw = '{:.2f}'.format(float(gyro['yaw']))
    accelerom = sense.get_accelerometer()
    accelerom_roll =  '{:.2f}'.format(float(accelerom['roll']))
    accelerom_pitch = '{:.2f}'.format(float(accelerom['pitch']))
    accelerom_yaw = '{:.2f}'.format(float(accelerom['yaw']))

    dir_name = data.strftime("dados_altafreq%d-%m-%y")
    if not os.path.exists(dir_folder+dir_name):
        os.mkdir(dir_folder+dir_name)
    filename_sense = data.strftime("senseHat_%Y_%m_%d_")+day_part
    sh_end = time.time()
    sh_timer = round(1000*(sh_end-sh_start), 2)
    dados_sense = [{"horario": horario, "elapsed(ms)":sh_timer,
        "umidade_relativa(percent)": umidade, "pressao(milliBar)":
        pressao,"compass_norte(deg)": compass,"gyro_roll(deg)":
        gyro_roll,"gyro_pitch(deg)": gyro_pitch,"gyro_yaw(deg)":
        gyro_yaw,"accelerometer_roll(deg)": accelerom_roll,"accelerometer_pith(deg)":
        accelerom_pitch,"accelerometer_yaw(deg)": accelerom_yaw}]
    file_loc_sense = dir_folder+dir_name+"/"+filename_sense+".csv"

    datapath.highfreq_writedata(file_loc_sense, header_sense, dados_sense, title_sense)
    if q_sh.empty():
        q_sh.put(dados_sense)

def read_SPI_flutuante_sensors():
    global spi_start
    global spi_end
    data = datetime.date.today()
    horario = datetime.datetime.now()

    day_part = get_daypart(horario)
    horario = horario.strftime('%H:%M:%S')

    #Desligar CS de todas placas
    pi.write(GPIO_TENSAO_P1P2,1)
    pi.write(GPIO_TENSAO_P3P4, 1)
    pi.write(GPIO_CURR, 1)
    pi.write(GPIO_IRR, 1)

    #Canais de tensão da associação P1 P2 --- definir taxa de ganho dos divisores
    pi.write(GPIO_TENSAO_P1P2, 0)
    ad7705.initChannel(CHN_AIN1)
    tensao_p1 = ((ad7705.readADResultRaw(CHN_AIN1)*ESCALA)-OFF_AD620_P1)*op_tp1
    ad7705.initChannel(CHN_AIN2)
    tensao_p2 = ad7705.readADResultRaw(CHN_AIN2)*op_tp2
    pi.write(GPIO_TENSAO_P1P2, 1)
    #Canais de tensão da associação P3 P4 --- definir taxa de ganho dos divisores
    pi.write(GPIO_TENSAO_P3P4, 0)
    ad7705.initChannel(CHN_AIN1)
    tensao_p3 = ((ad7705.readADResultRaw(CHN_AIN1)*ESCALA)-OFF_AD620_P3)*op_tp3
    ad7705.initChannel(CHN_AIN2)
    tensao_p4 = ad7705.readADResultRaw(CHN_AIN2)*op_tp4
    pi.write(GPIO_TENSAO_P3P4, 1)
    #Canal de correntes das associações --- definir ganho
    pi.write(GPIO_CURR, 0)
    ad7705.initChannel(CHN_AIN1)
    corrente_p1p2_v = ad7705.readADResultRaw(CHN_AIN1)*ESCALA
    corrente_p1p2 = (corrente_p1p2_v*ACS712_ESCALA)-ACS712_OFFSET
```

```
    ad7705.initChannel(CHN_AIN2)
    corrente_p3p4_v = ad7705.readADResultRaw(CHN_AIN2)*ESCALA
    corrente_p3p4 = (corrente_p3p4_v*ACS712_ESCALA)-ACS712_OFFSET
    pi.write(GPIO_CURR, 1)
    #Canal do piranômetro ---definir ganho e escala de acordo com a curva do sensor para
        definir irradiacao
    pi.write(GPIO_IRR, 0)
    ad7705.initChannel(CHN_AIN2)
    irradiacao = ((ad7705.readADResultRaw(CHN_AIN1)*op_irr)-OFF_AD620_IRR)*10
    pi.write(GPIO_IRR, 1)

    tensao_p1 = '{:.2f}'.format(float(tensao_p1))
    tensao_p2 = '{:.2f}'.format(float(tensao_p2))
    tensao_p3 = '{:.2f}'.format(float(tensao_p3))
    tensao_p4 = '{:.2f}'.format(float(tensao_p4))
    corrente_p1p2 = '{:.2f}'.format(float(corrente_p1p2))
    corrente_p3p4 = '{:.2f}'.format(float(corrente_p3p4))
    irradiacao = '{:.2f}'.format(float(irradiacao))

    dir_name = data.strftime("dados_altafreq%d-%m-%y")
    if not os.path.exists(dir_folder+dir_name):
        os.mkdir(dir_folder+dir_name)
    filename = data.strftime("SPI_flut_%Y_%m_%d_")+day_part
    spi_end = time.time()
    spi_timer = round(1000*(spi_end-spi_start), 2)
    dados = [
    {"horario": horario, "elapsed(ms)": spi_timer, "tensao_p1(V)": tensao_p1,
        "tensao_p2(V)": tensao_p2, "tensao_p3(V)": tensao_p3, "tensao_p4(V)": tensao_p4,
        "corrente_p1p2(A)": corrente_p1p2, "corrente_p3p4(A)": corrente_p3p4,
        "irradiacao(W/m2)": irradiacao}
    ]

    file_loc = dir_folder+dir_name+"/"+filename+".csv"

    datapath.highfreq_writedata(file_loc, header, dados, spi_title_flutuante)
    if q_spi.empty():
        q_spi.put(dados)

def read_SPI_fixo_sensors():
    global spi_start
    global spi_end
    data = datetime.date.today()
    horario = datetime.datetime.now()

    day_part = get_daypart(horario)

    horario = horario.strftime('%H:%M:%S')
    #Desligar CS todas placas
    pi.write(GPIO_TENSAO,1)
    pi.write(GPIO_IRR,1)

    #Canais de tensão --- definir taxa de ganho dos divisores
    #Ativar o CS
    pi.write(GPIO_TENSAO, 0)
    ad7705.initChannel(CHN_AIN1)
    tensao_p1 = ((ad7705.readADResultRaw(CHN_AIN1)*ESCALA)-
                 OFF_AD620_P1)*(1/G_AD620_P1)*(1/G_P25_P1)
    ad7705.initChannel(CHN_AIN2)
    tensao_p2 = ad7705.readADResultRaw(CHN_AIN2)*ESCALA*(1/G_P25_P2)
    #Desativar o CS
    pi.write(GPIO_TENSAO, 1)

    tensao_p1 = '{:.2f}'.format(float(tensao_p1))
    tensao_p2 = '{:.2f}'.format(float(tensao_p2))

    if(LER_IRR):
        pi.write(GPIO_IRR, 0)
        ad7705.initChannel(CHN_AIN1)
        irradiacao =
            ((ad7705.readADResultRaw(CHN_AIN1)*ESCALA)-OFF_AD620_IRR)*(1/G_AD260_IRR)
        #Desativar o CS
        pi.write(GPIO_IRR, 1)
        irradiacao = '{:.2f}'.format(float(irradiacao))
    else:
```

```
        irradiacao = "NaN"

    dir_name = data.strftime("dados_altafreq%d-%m-%y")
    if not os.path.exists(dir_folder+dir_name):
        os.mkdir(dir_folder+dir_name)
    filename = data.strftime("SPI_fixo_%Y_%m_%d_")+day_part
    spi_end = time.time()
    spi_timer = round(1000*(spi_end-spi_start), 2)
    dados = [
    {"horario": horario, "elapsed(ms)": spi_timer, "tensao_p1(V)": tensao_p1,
        "tensao_p2(V)": tensao_p2, "irradiacao(W/m2)": irradiacao}
    ]
    file_loc = dir_folder+dir_name+"/"+filename+".csv"
    datapath.highfreq_writedata(file_loc, header, dados, spi_title_fixo)
    if q_spi.empty():
        q_spi.put(dados)

def spi_read(is_earth, q):
    global spi_start
    global q_spi

    q_spi = q
    if is_earth == False:
        while True:
            spi_start = time.time()
            read_SPI_flutuante_sensors()
            #time.sleep(.01)
    else:
        while True:
            spi_start = time.time()
            read_SPI_fixo_sensors()
            #time.sleep(.01)
def sh_read(is_earth, q):
    global sh_start
    global q_sh

    q_sh = q
    if is_earth == False:
        while True:
            sh_start = time.time()
            time.sleep(.045)
            read_senseHat_sensors()
    else:
        pass

#Those are test functions
def main(args):
    spi_read()
    sh_read()

def testing():
    import multiprocessing as mp
    initialize(False)
    proc_spi = mp.Process(target=spi_read, args=(False,))
    proc_sh = mp.Process(target=sh_read, args=(False,))
    proc_spi.start()
    proc_sh.start()
    print('Press Ctrl-C to stop.')
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        proc_spi.terminate()
        proc_sh.terminate()
        os.system('sudo killall pigpiod')
        print('Program finished.')

if __name__ == '__main__':
    import sys
    testing()
    exit()
```

Código 13 – Código do módulo de dados

```
"""
File: datapath.py

Description: This file is responsible for anything related to the data in the project.
"""
#Libraries
import csv
import os
import shutil
import json
import pandas as pd
from datetime import datetime as date
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive

#Filenames params
lowfreq_filename = "dados_baixafreq"
highfreq_filename = r'dados_altafreq'
log_filename = "system_log"
data_dir = r'/dados/'
sensors_dir = r'/sensors_config/'

#Read a column from csv
def csv_readcol(file, column):
    cwd = os.getcwd()
    datapath = cwd+data_dir
    time_now = date.now()
    time_now = time_now.strftime('%d/%m/%y')
    time_now = time_now.replace('/','-')

    file = datapath+file+time_now+".csv"
    return pd.read_csv(file)[column].values

#Read a row from csv
def csv_getrows(file):
    cwd = os.getcwd()
    datapath = cwd+data_dir
    time_now = date.now()
    time_now = time_now.strftime('%d/%m/%y')
    time_now = time_now.replace('/','-')

    file = datapath+file+time_now+".csv"
    with open(file) as csv_file:
        csv_reader = csv.reader(csv_file, delimiter = ',')
        for row in csv_reader:
            return row

 #Create data dir to avoid bugs/crash
def initialize():
    cwd = os.getcwd()
    datapath = cwd+data_dir
    if not os.path.exists(datapath):
        os.mkdir(datapath)

#Upload routine
def data_upload():
    print('Upload routine started, please wait...')
    cwd = os.getcwd()
    time_now = date.now()
    time_now = time_now.strftime('%d/%m/%y')
    time_now = time_now.replace('/','-')

    datapath = cwd+data_dir
    lowfreq_file = datapath+lowfreq_filename+time_now+".csv"
    log_file = datapath+log_filename+time_now+".txt"
    highfreq_dir = datapath+highfreq_filename+time_now+r'/'
    data_folder = datapath+r'dados'+time_now
    os.system(f"mkdir {data_folder}")
    os.system(f"mv {lowfreq_file} {log_file} {highfreq_dir} {data_folder}")
    #os.system(f"mv {lowfreq_file} {data_folder}")

    zipfile = "dados"+time_now+".zip"
```

```
    gauth=GoogleAuth()
    gauth.LocalWebserverAuth()  # Cria um servidor da web local e gerencia a
        autenticação automática
    drive = GoogleDrive(gauth)
    drive_file = drive.CreateFile({'parents':
        [{'id':'1A3uP92xXJKLdMz4-Mhz0APd5W9QFXPSK'}],'title':
        zipfile})#'dados17-09-21.zip
    #id = folder id at GDrive

    zipfile = "dados"+time_now
    shutil.make_archive(datapath+zipfile, 'zip', data_folder)
    zipfile = zipfile+".zip"
    drive_file.SetContentFile(datapath+zipfile)
    drive_file.Upload()
    print('Uploaded!')
    os.system(f"rm -r {data_folder}")
    return True

#Write csv data at corresponding rows
def sensors_writedata(fieldNames,B):
    cwd = os.getcwd()
    hora=date.now()
    c=hora.strftime('%d/%m/%y')
    c= c.replace('/','-')
    name=lowfreq_filename+c+'.csv'
    fileName=cwd+data_dir+name
    if os.path.exists(fileName)==False:
        with open(fileName,'w',newline='') as csvfile:
                writer = csv.DictWriter(csvfile, fieldnames=fieldNames)
                writer.writeheader()
                writer.writerow(B)
                csvfile.close()
    else:
        with open(fileName,'a',newline='') as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=fieldNames)
            writer.writerow(B)
            csvfile.close()

#Same as previous, but for highfreq
def highfreq_writedata(filename, headers, data, title):
    if (os.path.isfile(filename)):
        with open(filename, "a") as csvfile:
            arquivo = csv.DictWriter(csvfile, fieldnames=headers)
            arquivo.writerows(data)
    else:
        with open(filename, "a") as csvfile:
            arquivo = csv.writer(csvfile, delimiter=';', lineterminator='\r')
            arquivo.writerow([title])
            arquivo.writerow([' '])
            arquivo = csv.DictWriter(csvfile, fieldnames=headers)
            arquivo.writeheader()
            arquivo.writerows(data)
    csvfile.close()

#Removes sensors from list upon failure (json).
def sensor_remove(filename, sensor):
    cwd = os.getcwd()
    file = cwd+sensors_dir+filename
    with open(file, 'r+') as f:
        data = json.load(f)
        data[sensor] = 1
        f.seek(0)
        json.dump(data, f, indent=4)
        f.truncate()
        f.close()

'''
#Removes sensors from list upon failure (csv row delete).
def sensor_remove(filename, sensor):
    lines = list()
    cwd = os.getcwd()
    file = cwd+sensors_dir+filename
    with open(file, 'r') as readFile:
```

```
        reader = csv.reader(readFile)
        for row in reader:
            lines.append(row)
            for field in row:
                if field == sensor:
                    lines.remove(row)

    with open(file, 'w') as writeFile:
        writer = csv.writer(writeFile)
        writer.writerows(lines)
'''
#Log writer
def log_write(info):
    cwd = os.getcwd()
    time_now = date.now()
    time_now = time_now.strftime('%d/%m/%y')
    time_now = time_now.replace('/','-')

    datapath = cwd+data_dir
    log_file = datapath+log_filename+time_now+".txt"
    current_time = '['+date.now().strftime("%H:%M:%S")+'] - '
    with open(log_file, 'a') as file:
        file.write(current_time+info+'\n')
        file.close()

#Json reader
def json_read(file):
    cwd = os.getcwd()
    file = cwd+sensors_dir+file
    with open(file, 'r+') as f:
        data = json.load(f)
        f.close()
    return data

#Json editor
def json_replace(file, key, value):
    cwd = os.getcwd()
    file = cwd+sensors_dir+file
    with open(file, 'r+') as f:
        data = json.load(f)
        data[key] = value
        f.seek(0)
        json.dump(data, f, indent=4)
        f.truncate()
        f.close()

if __name__ == "__main__":
  data_upload()
```

Código 14 – Código do módulo de interface gráfica

```
"""
File: gui.py

Description: This file is responsible for the graphical user interface (GUI).

"""

import tkinter as tk
import tkinter.font as tkFont
import matplotlib.pyplot as plt
import datapath as data
from tkinter import ttk

#Setting up windows
w_def = tk.Tk() #Default window
w_graphs_combo = [] #Graphs window combo
w_def.title("PV Monitor")

width=800
height=600
screenwidth = w_def.winfo_screenwidth()
screenheight = w_def.winfo_screenheight()
```

```
alignstr = '%dx%d+%d+%d' % (width, height, (screenwidth - width) / 2, (screenheight -
    height) / 2)

w_def.geometry(alignstr)
w_def.resizable(width=False, height=False)

#Global variables
labels = {} #Vetor que guarda ids
list = [] #Vetor que guarda infos
v_labels = [] #Vetor que guarda labels objects
v_list = [] #Vetor que guarda list objects
legends = []

DIST_LALI = 150
DIST_PARAMS = 220

def exit(root):
    from main import q_except
    root.destroy()
    w_graphs.destroy()
    q_except.put(True)
    #system_shutdown()

def plot_grafico():
    global legends
    grandeza = w_graphs_combo.get()
    legends.append(grandeza)

    x = data.csv_readcol(data.lowfreq_filename, 'Hora')
    y = data.csv_readcol(data.lowfreq_filename, grandeza)
    plt.plot(x, y)
    plt.title('Curves plot')
    plt.xlabel("Time")
    plt.ylabel("Value")

    num_ticks = 6
    tick_step = len(x) // (num_ticks - 1)
    x_ticks = x[::tick_step]
    plt.xticks(x_ticks)
    plt.legend(legends)
    plt.show()
    legends = []

def invoke_w_graphs():
    global w_graphs_combo
    global w_graphs
    #global variables
    w_graphs = tk.Tk() #Graphs window
    #Criando janela de graficos
    w_graphs.title("Plot stat")
    #Frame
    frame = ttk.Frame(w_graphs)
    frame.pack(padx=10, pady=10)
    #Dropdawn menu
    label = ttk.Label(frame, text="Choose a parameter")
    label.grid(row=0, column=0, padx=5, pady=5)
    variables = data.csv_getrows(data.lowfreq_filename)
    w_graphs_combo = ttk.Combobox(frame, values=variables)
    w_graphs_combo.grid(row=0, column=1, padx=5, pady=5)

    # Botão para plotar o gráfico
    botao = ttk.Button(frame, text="Plot", command = plot_grafico)
    botao.grid(row=1, columnspan=2, padx=5, pady=5)
    w_graphs.mainloop()

def send_console(msg, consoleid):
    consoleid.insert(tk.END, msg)

def clear_console(consoleid):
    consoleid.delete(1.0, tk.END)

def build_element(list, label, posx, posy, text):
    ft = tkFont.Font(family='Times',size=10)
    label["font"] = ft
```

```
    label["fg"] = "#333333"
    label["justify"] = "center"
    label["text"] = text
    label.place(x=posx,y=posy,width=150,height=25)

    posx += DIST_LALI #Label-list

    list["borderwidth"] = "1px"
    list["font"] = ft
    list["fg"] = "#333333"
    list["justify"] = "left"
    list.place(x=posx,y=posy,width=60,height=25)

    posx -= DIST_LALI
    posy += 30 #List-Label
    return posx, posy

def initialize(root, variables):
    #Criando janela principal
    #Usar um dicionario com id do vetor.
    #Usar o id pra escrever nas labels.
    global labels
    global list
    global console

    posx = 10 #Passo 70: Label-List. Passo 90: List-Label
    posy = 10 #Passo 30, 1 pra outro.
    for i in range(0, len(variables)): #Label e listbox para cada var.
        v_list.append(tk.Listbox(root)) #Vetor de listboxes
        v_labels.append(tk.Label(root)) #Vetor de labels
        labels.update({variables[i]:i}) #Vetor que guarda o id para as listboxes

        posx, posy = build_element(v_list[i], v_labels[i], posx, posy, variables[i])
        if posy > 490:
            posy = 10
            posx += DIST_PARAMS
        if posx > 620:
            break

    ft = tkFont.Font(family='Times',size=10)
    console = tk.Text(root)
    console["font"] = tkFont.Font(family='Times',size=10)
    console.place(x=480,y=400,width=300,height=112)

    end_button = tk.Button(root, text = 'End', command = lambda:exit(root))
    end_button["bg"] = "#f0f0f0"
    end_button["font"] = ft
    end_button["fg"] = "#000000"
    end_button["justify"] = "center"
    end_button.place(x=340,y=550,width=70,height=25)

    graphs_button = tk.Button(root, text = 'Stats', command = lambda:invoke_w_graphs())
    graphs_button["bg"] = "#f0f0f0"
    graphs_button["font"] = ft
    graphs_button["fg"] = "#000000"
    graphs_button["justify"] = "center"
    graphs_button.place(x=700,y=100,width=70,height=25)

def post_information(variables, data, labels):
    #data = queue.get()
    for i in variables:
        v_list[labels[i]].delete(0)
        v_list[labels[i]].insert(0, data[i])

def start():
    w_def.mainloop()
```

Código 15 – Código do módulo Watchdog

```
"""
File: watchdog.py

Description: This file is responsible for monitoring the systems and
```

```
            security measures.

"""
import hardware as hardw
import datapath as data
#import shspi as ss
import threading as thr
import time
import os
import signal
from datetime import datetime as date

#Queues
q_watchdog = 0
q_except = 0
q_spi = 0
q_sh = 0

#System params
modbus_devices = 0
ow_devices = 0
variables = 0
pid_main = 0
pid_sh = 0
pid_spi = 0
pid_father = 0

#Sensor fault
faulty_counter = {}
previous_faulty = []
current_reading = []
spi_read = []
sh_read = []
reset_counter = 0
reset_reason = []

#Time params (h, m)
t_reboot = []
t_sinaliz_on = []
t_sinaliz_off = []

#Thresholds
thr_pump1 = 0.0
thr_pump2 = 0.0
thr_pump3 = 0.0
thr_vent1 = 0.0
thr_vent2 = 0.0

is_on = {'bomba1':False, 'bomba2':False, 'bomba3':False, 'sinalizador':False,
    'vent_1':False, 'vent_2':False}

#Gets time
def get_time():
  hour_str = date.now()
  hour_str = hour_str.strftime('%H:%M:%S')
  return hour_str

#Get most frequent element of a list
def most_frequent(List):
    return max(set(List), key = List.count)

#Uploads file
def upload():
    data.log_write('Upload routine started.')
    data.data_upload()

#Idle function
def idle():
    data.log_write('System is now in idle mode. No data is being monitored.')
    data.log_write('System will reboot in 1 hour.')
    upload()
    time.sleep(3600)

#Reboot function
```

```
def system_reboot():
    from main import system_cleanup
    os.kill(pid_main, signal.SIGKILL)
    os.kill(pid_spi, signal.SIGKILL)
    if not hardw.is_earth:
        os.kill(pid_sh, signal.SIGKILL)
    idle()
    system_cleanup()
    os.system('systemctl reboot')

#General monitor
def misc_monitor():
    global spi_read
    global sh_read

    sh_check = False
    while True:
        try:
            t_current = get_time()
            t_current = t_current.split(':')
            hour = int(t_current[0])
            min = int(t_current[1])

            if hour == t_reboot[0] and min == t_reboot[1]:
                system_reboot()

            if not hardw.is_earth:
                #Fieldlogger termopar 7 bomba cortina
                if float(current_reading['FL_TC7 (°C)']) >= thr_pump2 and not
                    is_on['bomba2']:
                    hardw.GPIO.output(hardw.bomba2, hardw.GPIO.LOW)
                    is_on['bomba2'] = True
                    data.log_write(f'Curtain pump turned on.')

                if float(current_reading['FL_TC7 (°C)']) < thr_pump2 and is_on['bomba2']:
                    hardw.GPIO.output(hardw.bomba2, hardw.GPIO.HIGH)
                    is_on['bomba2'] = False
                    data.log_write(f'Curtain pump turned off.')

                #Fieldlogger termopar 3 bomba backsheet
                if float(current_reading['FL_TC3 (°C)']) >= thr_pump1 and not
                    is_on['bomba1']:
                    hardw.GPIO.output(hardw.bomba1, hardw.GPIO.LOW)
                    is_on['bomba1'] = True
                    data.log_write(f'Backsheet pump turned on.')

                if float(current_reading['FL_TC3 (°C)']) < thr_pump1 and is_on['bomba1']:
                    hardw.GPIO.output(hardw.bomba1, hardw.GPIO.HIGH)
                    is_on['bomba1'] = False
                    data.log_write(f'Backsheet pump turned off.')

                #Bomba trocador
                #Am8t - Termopar 1 = TC9 (Face inferior painel inferior)
                #Am8t - Termopar 2 = TC10 (Trocador de calor)
                if abs(float(current_reading['AM_TC1
                    (°C)'])-float(current_reading['AM_TC2 (°C)'])) >= thr_pump3 and not
                    is_on['bomba3']:
                    hardw.GPIO.output(hardw.bomba3, hardw.GPIO.LOW)
                    is_on['bomba3'] = True
                    data.log_write(f'Heat exchanger pump turned on.')

                if abs(float(current_reading['AM_TC1
                    (°C)'])-float(current_reading['AM_TC2 (°C)'])) < thr_pump3 and
                    is_on['bomba3']:
                    hardw.GPIO.output(hardw.bomba3, hardw.GPIO.HIGH)
                    is_on['bomba3'] = False
                    data.log_write(f'Heat exchanger pump turned off.')

                if (hour >= t_sinaliz_on[0] or hour < t_sinaliz_off[0]) and not
                    is_on['sinalizador']:
                    hardw.GPIO.output(hardw.sinalizador, hardw.GPIO.LOW)
                    is_on['sinalizador'] = True
                    data.log_write(f'Signal lights turned on.')
```

```
                if hour == t_sinaliz_off[0] and is_on['sinalizador']:
                    hardw.GPIO.output(hardw.sinalizador, hardw.GPIO.HIGH)
                    is_on['sinalizador'] = False
                    data.log_write(f'Signal lights turned off.')


            if spi_read and sh_read and (sh_check == False):
                os.system('clear')
                for key, value in is_on.items():
                    print(key+':', value)
                print()

                spi_read_temp = spi_read[0]
                spi_read_temp.pop('horario')
                spi_read_temp.pop('elapsed(ms)')

                for key, value in spi_read_temp.items():
                    print(key+':', value)
                print()

                if not hardw.is_earth:
                    sh_read_temp = sh_read[0]
                    sh_read_temp.pop('horario')
                    sh_read_temp.pop('elapsed(ms)')
                    for key, value in sh_read_temp.items():
                        print(key+':', value)
                    print()

                sh_check = True

            time.sleep(8)
            sh_check = False
        except:
            #This prevents thread from exiting. It will never stop unless proc is killed
            pass

#Vent monitor
def vent_monitor():
    global is_on
    while True:
        try:
            pi_temp = hardw.get_cpu_temp()
            #print(pi_temp)
            if pi_temp >= thr_vent1 and not is_on['vent_1']:
                hardw.GPIO.output(hardw.vent_1, hardw.GPIO.LOW)
                is_on['vent_1'] = True
                data.log_write(f'CPU temperature is {pi_temp} degrees. Fan is now on.')

            if pi_temp < thr_vent1 and is_on['vent_1']:
                hardw.GPIO.output(hardw.vent_1, hardw.GPIO.HIGH)
                is_on['vent_1'] = False

            if not hardw.is_earth:
                #Controlador 2
                if float(current_reading['TC2 (°C)']) >= thr_vent2 and not
                    is_on['vent_2']:
                    hardw.GPIO.output(hardw.vent_2, hardw.GPIO.LOW)
                    is_on['vent_2'] = True
                    data.log_write(f'Controller temperature higher than 48 degrees. Fan
                        is now on.')

                if float(current_reading['TC2 (°C)']) < thr_vent2 and is_on['vent_2']:
                    hardw.GPIO.output(hardw.vent_2, hardw.GPIO.HIGH)
                    is_on['vent_2'] = False
            '''
            #Fixed system needs conection.
            else:
                if float(current_reading['TC3 (°C)']) >= 48.0 and not is_on['vent_2']:
                    hardw.GPIO.output(hardw.vent_2, hardw.GPIO.LOW)
                    is_on['vent_2'] = True

                if float(current_reading['TC3 (°C)']) < 48.0 and is_on['vent_2']:
                    hardw.GPIO.output(hardw.vent_2, hardw.GPIO.HIGH)
                    is_on['vent_2'] = False
```

```
            '''
            time.sleep(0.5)
        except:
            pass #This thread should also not exit.

#Check to see if sensors are okay
def check():
    global q_watchdog
    global reset_counter
    global reset_reason
    global spi_read
    global current_reading
    global sh_read

    previous_faulty = []
    reset_counter = 0
    reset = False
    time.sleep(10)
    while True:
        if q_watchdog.empty():
            time.sleep(0.5)
            continue
        else:
            current_reading = q_watchdog.get() #Gets reading and faulty sensors
            faulty = q_watchdog.get()
            if not q_spi.empty():
                spi_read = q_spi.get()
            if not hardw.is_earth and not q_sh.empty():
                sh_read = q_sh.get()

        if not faulty:
            for key, value in faulty_counter.items():
                faulty_counter.update({key:0})

        for i in faulty:
            if i in previous_faulty:
                counter = faulty_counter[i]+1
                faulty_counter.update({i:counter}) #Increments if last reading was
                    faulty, too.
            else:
                faulty_counter.update({i:1})

        previous_faulty = faulty

        for key, value in faulty_counter.items():
            #print(key, value)
            if value > 4:# 4
                reset_reason.append(key)
                reset = True

        if reset == True:
            faulty_sensor = most_frequent(reset_reason)
            if reset_counter > 2: #2
                data.log_write(f'Sensor {faulty_sensor} fails repeateadly. It won\'t be
                    read anymore.')
                reset_counter = 0
                reset = False
                #print('Needs to remove sensor!')
                #print(most_frequent(reset_reason))
                data.sensor_remove('sensors_faulty.json', faulty_sensor)
                reset_reason = []
                q_except.put(0)
                for key, value in faulty_counter.items():
                    faulty_counter.update({key:0})
            else:
                data.log_write(f'Reloading Modbus due to faulty sensor: {faulty_sensor}')
                #print('Modbus resetting because of faulty component:', faulty_sensor)
                hardw.modbus_hard_reset()
                reset_counter += 1
                for key, value in faulty_counter.items():
                    faulty_counter.update({key:0})
                reset = False
                #time.sleep(25)
```

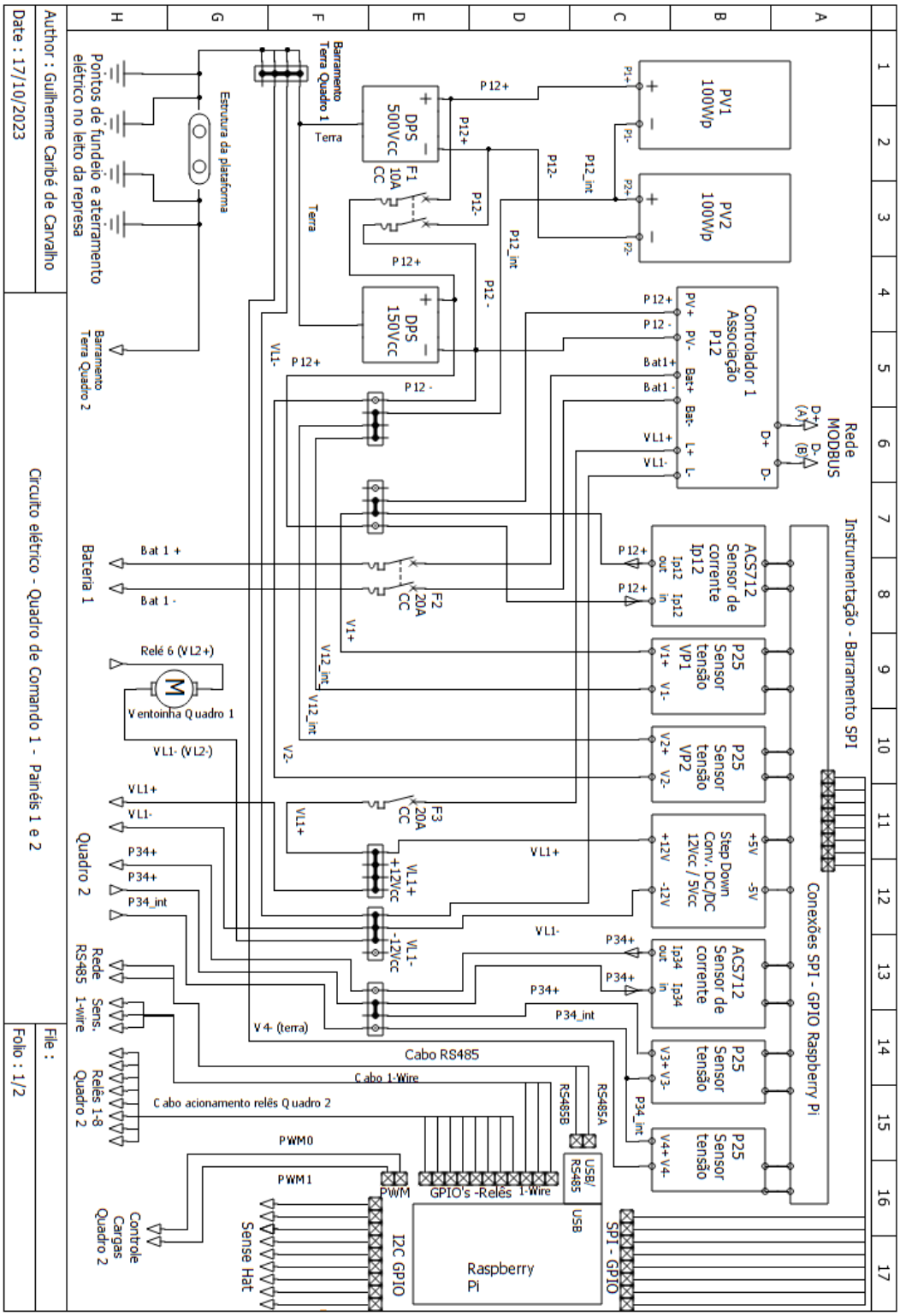
```
        #print('Faulty:', faulty)
        #print('Previous faulty:', previous_faulty)

        q_watchdog.task_done()
        q_watchdog.task_done()
        time.sleep(10)

#Loads important params
def initialize():
    global pid_main
    global pid_sh
    global pid_spi

    global t_reboot
    global t_sinaliz_on
    global t_sinaliz_off

    global thr_pump1
    global thr_pump2
    global thr_pump3
    global thr_vent1
    global thr_vent2

    modbus_devices = q_watchdog.get()
    ow_devices = q_watchdog.get()
    variables = q_watchdog.get()
    pid_main = q_watchdog.get()
    pid_spi = q_watchdog.get()

    if not hardw.is_earth:
        pid_sh = q_watchdog.get()
        q_watchdog.task_done()

    for device in modbus_devices:
        faulty_counter.update({device['nome']:0})

    q_watchdog.task_done()
    q_watchdog.task_done()
    q_watchdog.task_done()
    q_watchdog.task_done()
    q_watchdog.task_done()

    thresholds = data.json_read('sys_thresholds.json')
    t_reboot.extend(thresholds['t_reboot'].split(':'))
    t_sinaliz_on.extend(thresholds['t_lights_on'].split(':'))
    t_sinaliz_off.extend(thresholds['t_lights_off'].split(':'))
    t_reboot = [int(a) for a in t_reboot]
    t_sinaliz_on = [int(a) for a in t_sinaliz_on]
    t_sinaliz_off = [int(a) for a in t_sinaliz_off]
    thr_pump1 = thresholds['thr_pump1']
    thr_pump2 = thresholds['thr_pump2']
    thr_pump3 = thresholds['thr_pump3']
    thr_vent1 = thresholds['thr_vent1']
    thr_vent2 = thresholds['thr_vent2']

#Set system up
def main(q1, q3, q4, q5):
    global q_watchdog
    global q_except
    global q_spi
    global q_sh

    q_watchdog = q1
    q_except = q3
    q_spi = q4
    q_sh = q5
    thr_miscm = thr.Thread(target = misc_monitor)
    thr_check = thr.Thread(target = check)
    thr_vents = thr.Thread(target = vent_monitor)
    initialize()

    thr_vents.start()
    thr_check.start()
    thr_miscm.start()
```

```
    data.log_write('Watchdog module was started and is monitoring the system.')
```

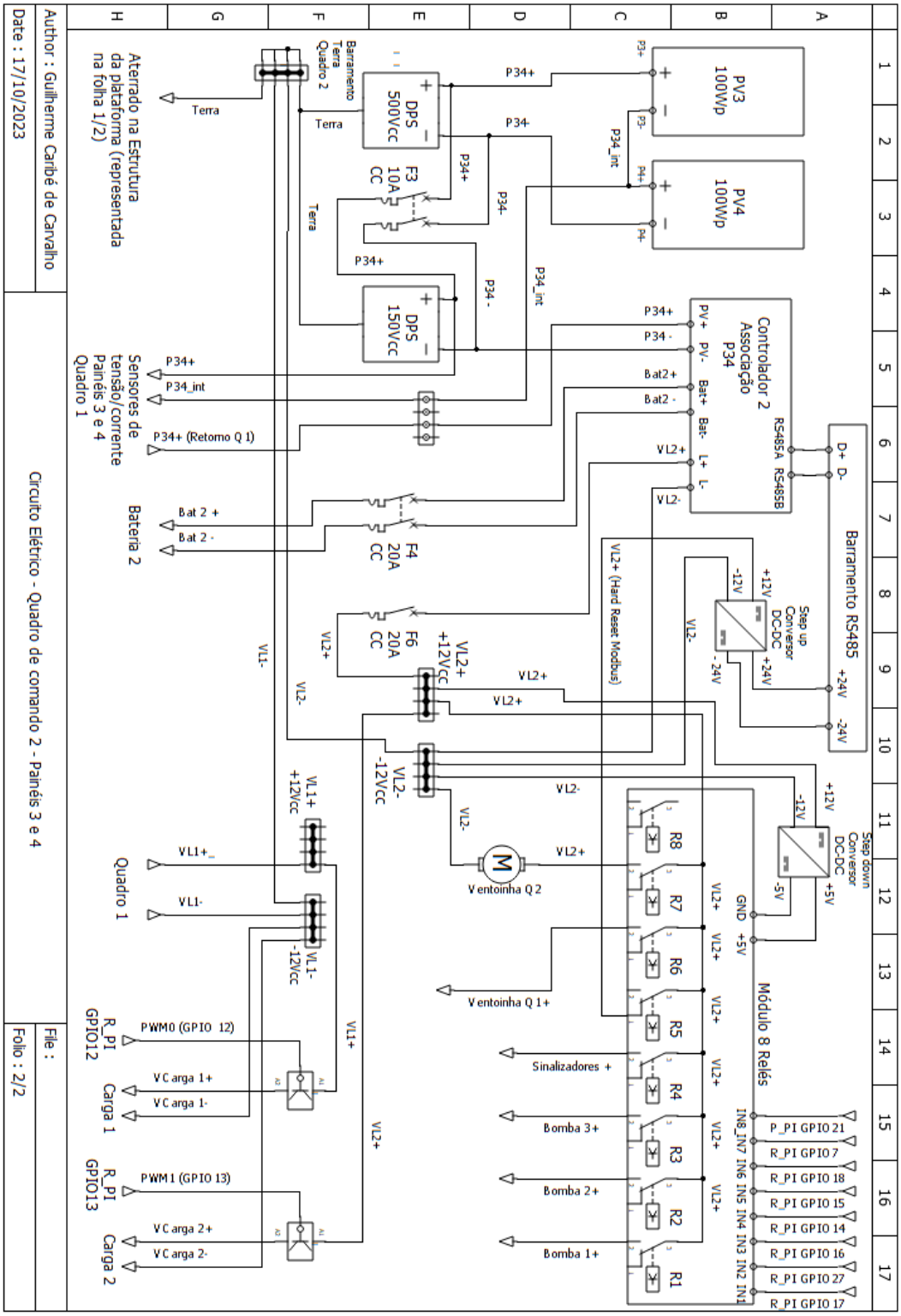
# Anexos

# Anexo A – Diagramas de terceiros

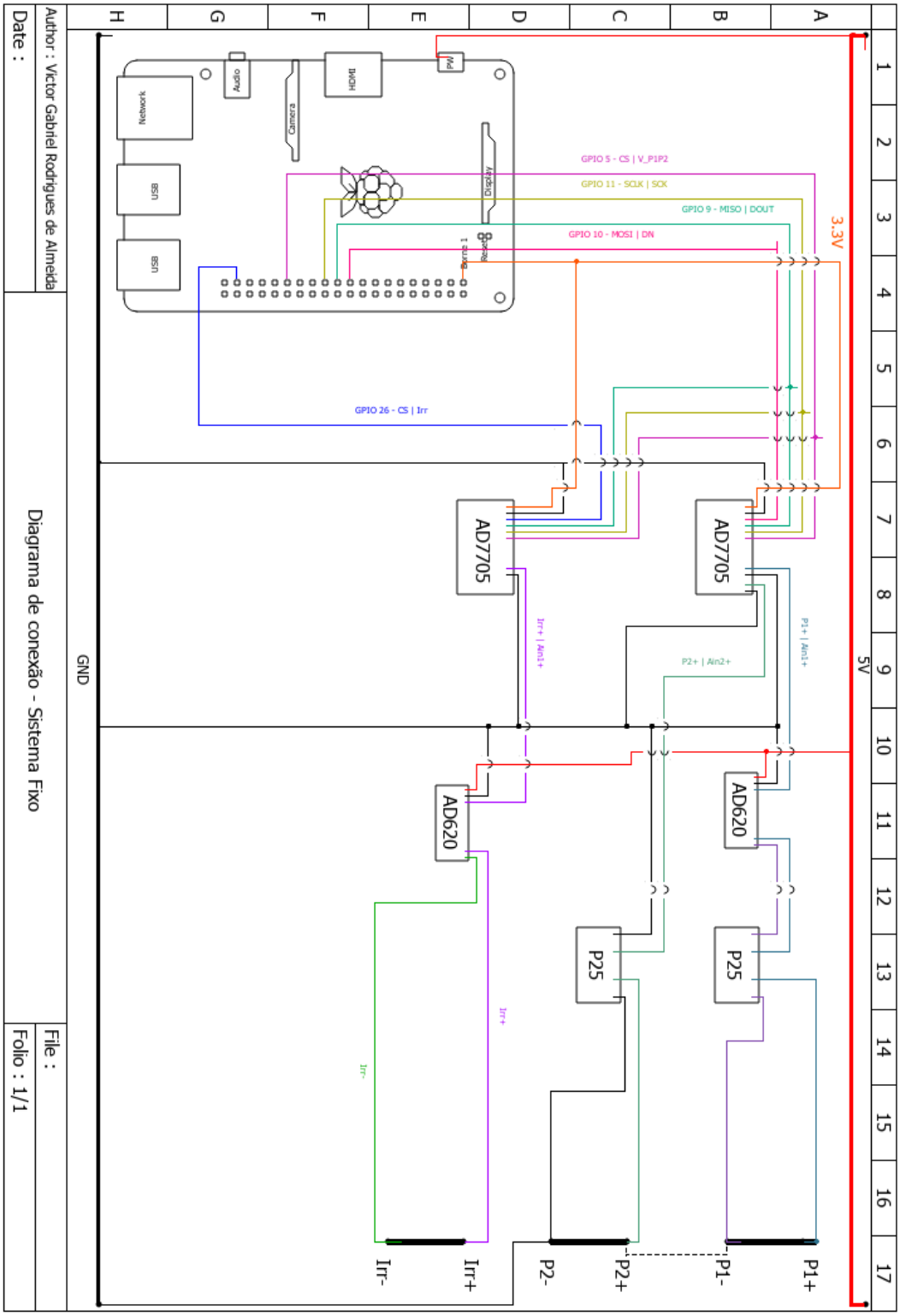
## A.1 Quadro de comando 1



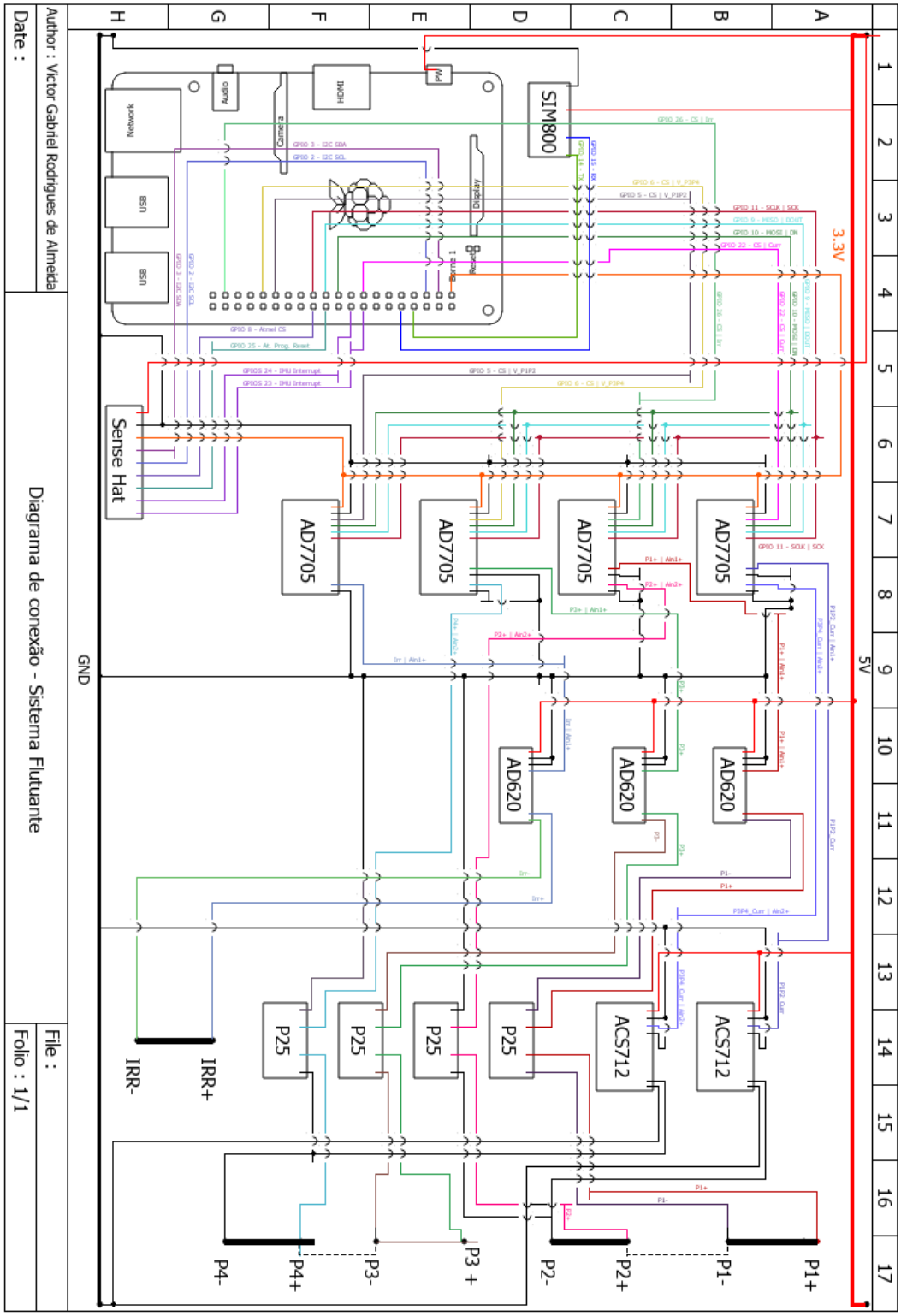
## A.2 Quadro de comando 2



## A.3 Diagrama unifilar do sistema fixo



## A.4 Diagrama unifilar do sistema flutuante



# Anexo B – Códigos de terceiros

## B.1 Driver AD770X

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
#  sans titre.py
#
#  Copyright 2016 belese <belese@belese-VPCEB3S1E>
#
#  This program is free software; you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation; either version 2 of the License, or
#  (at your option) any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
#  along with this program; if not, write to the Free Software
#  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
#  MA 02110-1301, USA.
#
#
#adapted form arduino library :
#https://github.com/kerrydwong/AD770X

import spidev

REG_CMM = 0x0 #communication register 8 bit
REG_SETUP = 0x1 #setup register 8 bit
REG_CLOCK = 0x2 #clock register 8 bit
REG_DATA = 0x3 #data register 16 bit, contains conversion result
REG_TEST = 0x4 #test register 8 bit, POR 0x0
REG_NOP = 0x5 #no operation
REG_OFFSET = 0x6 #offset register 24 bit
REG_GAIN = 0x7 # gain register 24 bit

#channel selection for AD7706 (for AD7705 use the first two channel definitions)
#CH1 CH0
CHN_AIN1 = 0x0 #AIN1; calibration register pair 0
CHN_AIN2 = 0x1 #AIN2; calibration register pair 1
CHN_COMM = 0x2 #common; calibration register pair 0
CHN_AIN3 = 0x3 #AIN3; calibration register pair 2

#output update rate
#CLK FS1 FS0
UPDATE_RATE_20 = 0x0 # 20 Hz
UPDATE_RATE_25 = 0x1 # 25 Hz
UPDATE_RATE_100 = 0x2 # 100 Hz
UPDATE_RATE_200 = 0x3 # 200 Hz
UPDATE_RATE_50 = 0x4 # 50 Hz
UPDATE_RATE_60 = 0x5 # 60 Hz
UPDATE_RATE_250 = 0x6 # 250 Hz
UPDATE_RATE_500 = 0x7 # 500 Hz

#operating mode options
#MD1 MD0
MODE_NORMAL = 0x0 #normal mode
MODE_SELF_CAL = 0x1 #self-calibration
MODE_ZERO_SCALE_CAL = 0x2 #zero-scale system calibration, POR 0x1F4000, set FSYNC high
    before calibration, FSYNC low after calibration
MODE_FULL_SCALE_CAL = 0x3 #full-scale system calibration, POR 0x5761AB, set FSYNC high
    before calibration, FSYNC low after calibration
```

```
#gain setting
GAIN_1 = 0x0
GAIN_2 = 0x1
GAIN_4 = 0x2
GAIN_8 = 0x3
GAIN_16 = 0x4
GAIN_32 = 0x5
GAIN_64 = 0x6
GAIN_128 = 0x7

UNIPOLAR = 0x0
BIPOLAR = 0x1

CLK_DIV_1 = 0x0 #Clock divider. 0x1 enables it!
CLK_DIV_2 = 0x2

MODE = 0b11 #SPI_CPHA | SPI_CPOL
BITS = 8 #Bits per word
SPEED = 50000 #Transmission speed
DELAY = 10

ESCALA = 5/65535
#slp_time = 0.001

class AD770X():
    def __init__(self,bus=0,device=0) :
        self.spi = spidev.SpiDev()
        self.spi.open(bus, device)
        self.spi.max_speed_hz = SPEED
        self.spi.mode = 0b11
        self.spi.bits_per_word = BITS
        self.reset()

    #AD channel config
    def initChannel(self,channel,clkDivider=CLK_DIV_1,polarity=BIPOLAR,gain=GAIN_1,
                    updRate=UPDATE_RATE_200) :
        self.setNextOperation(REG_CLOCK, channel, 0)
        self.writeClockRegister(0, clkDivider, updRate)

        self.setNextOperation(REG_SETUP, channel, 0)
        self.writeSetupRegister(MODE_SELF_CAL, gain, polarity, 0, 0)

        while not self.dataReady(channel) :
            pass

    def setNextOperation(self,reg,channel,readWrite) :
        r = reg << 4 | readWrite << 3 | channel
        self.spi.xfer([r])

    '''
    Clock Register
       7       6        5         4         3         2       1       0
    ZERO(0) ZERO(0) ZERO(0) CLKDIS(0) CLKDIV(0) CLK(1) FS1(0) FS0(1)

    CLKDIS: master clock disable bit
    CLKDIV: clock divider bit
    '''
    def writeClockRegister(self,CLKDIS,CLKDIV,outputUpdateRate) :
        r = CLKDIS << 4 | CLKDIV << 3 | outputUpdateRate

        #r &= ~(1 << 2); # clear CLK
        r &= ~(0 << 2); # sets CLK
        self.spi.xfer([r])

    '''
    Setup Register
      7      6      5      4      3       2       1       0
    MD10) MD0(0) G2(0) G1(0) G0(0) B/U(0) BUF(0) FSYNC(1)
    '''
    def writeSetupRegister(self,operationMode,gain,unipolar,buffered,fsync) :
        r = operationMode << 6 | gain << 3 | unipolar << 2 | buffered << 1 | fsync
        self.spi.xfer([r])
```

```
    #16 bits word
    def readADResult(self) :
        b1 = self.spi.xfer([0x0])[0]
        b2 = self.spi.xfer([0x0])[0]

        r = int(b1 << 8 | b2)

        return r

    def readADResultRaw(self,channel) :
        while not self.dataReady(channel) :
            pass
        self.setNextOperation(REG_DATA, channel, 1)

        return self.readADResult()

    def dataReady(self,channel) :
        self.setNextOperation(REG_CMM, channel, 1)
        b1 = self.spi.xfer([0x0])[0]
        return (b1 & 0x80) == 0x0

    def reset(self) :
        for i in range(100) :
            self.spi.xfer([0xff])
```

Código 16 – AD770X.py