



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO DE TECNOLÓGICO
CURSO DE POSGRADUAÇÃO EM ENGENHARIA AMBIENTAL

Fernando Campo García

Título do trabalho: Subtítulo (se houver)

Florianópolis
2023

Fernando Campo García

Título do trabalho: Subtítulo (se houver)

Tese de Doutorado do Curso de Posgraduação em Engenharia Ambiental do Centro de Tecnológico da Universidade Federal de Santa Catarina para a obtenção do título de Doutor em Engenharia Ambiental.

Orientador: Prof. Leonardo Hoinaski, Dr.

Coorientador: Prof. Alejandro Rafael García Ramírez, Dr.

Florianópolis
2023

Ficha de identificação da obra

A ficha de identificação é elaborada pelo próprio autor.

Orientações em:

<http://portalbu.ufsc.br/ficha>

Fernando Campo García

Título do trabalho: Subtítulo (se houver)

Este Tese de Doutorado foi julgado adequado para obtenção do Título de “Doutor em Engenharia Ambiental” e aprovado em sua forma final pelo Curso de Posgraduação em Engenharia Ambiental.

Florianópolis, 15 de Dezembro de 2023.

Prof. XXXXXX, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Leonardo Hoinaski, Dr.
Orientador

Prof.(a) xxxx, Dr(a).
Avaliador(a)
Instituição xxxx

Prof.(a) xxxx, Dr(a).
Avaliador(a)
Instituição xxxx

Este trabalho é dedicado aos meus colegas de classe e aos meus queridos pais.

AGRADECIMENTOS

Inserir os agradecimentos aos colaboradores à execução do trabalho.

XX.

“Texto da Epígrafe.

Citação relativa ao tema do trabalho.

É opcional. A epígrafe pode também aparecer

na abertura de cada seção ou capítulo.

Deve ser elaborada de acordo com a NBR 10520.”

(Autor da epígrafe, ano)

RESUMO

No resumo são ressaltados o objetivo da pesquisa, o método utilizado, as discussões e os resultados com destaque apenas para os pontos principais. O resumo deve ser significativo, composto de uma sequência de frases concisas, afirmativas, e não de uma enumeração de tópicos. Não deve conter citações. Deve usar o verbo na voz ativa e na terceira pessoa do singular. O texto do resumo deve ser digitado, em um único bloco, sem espaço de parágrafo. O espaçamento entre linhas é simples e o tamanho da fonte é 12. Abaixo do resumo, informar as palavras-chave (palavras ou expressões significativas retiradas do texto) ou, termos retirados de thesaurus da área. Deve conter de 150 a 500 palavras. O resumo é elaborado de acordo com a NBR 6028.

Palavras-chave: Palavra-chave 1. Palavra-chave 2. Palavra-chave 3.

ABSTRACT

Resumo traduzido para outros idiomas, neste caso, inglês. Segue o formato do resumo feito na língua vernácula. As palavras-chave traduzidas, versão em língua estrangeira, são colocadas abaixo do texto precedidas pela expressão “Keywords”, separadas por ponto.

Keywords: Keyword 1. Keyword 2. Keyword 3.

LISTA DE FIGURAS

Figura 1 – Conjunto de bibliotecas utilizadas para <i>firmware</i> dos dispositivos CLEAN	27
Figura 2 – Diagramas de classes do pacote <i>Hardware Interfaces</i>	28
Figura 3 – Diagrama de classes do Módulo <i>Drivers</i>	29
Figura 4 – Diagramas de classes do módulo <i>Sensors</i>	31
Figura 5 – Diagrama de classes do pacote Data	35
Figura 6 – Processo de leitura de uma variável	36
Figura 7 – Aplicação web Renovar	37
Figura 8 – Estrutura da aplicação <i>Web Renovar</i>	37
Figura 9 – Entidades do banco de dados Renovar	38
Figura 10 – Camadas da aplicação <i>back-end</i> Renovar	40
Figura 11 – Classes de acesso ao dados da aplicação <i>back-end</i> Renovar	41
Figura 12 – Aplicação <i>front-end</i> da plataforma web Renovar	41
Figura 13 – Painéis da aplicação <i>front-end</i> de Renovar	42
Figura 14 – Painel de análise de dados da aplicação <i>back-end</i> Renovar	42
Figura 15 – Estrutura principal dos dispositivos. a) Medidor de gases fixo, e b) medidor móvel	43
Figura 16 – Ilustrações das versões (a) fixa e (b) móvel dos dispositivos de monitoramento	43
Figura 17 – A placa CLEAN Arduino Mega: (a) projeto PCB, (b) vista superior da placa, (c) vista inferior da placa.	44
Figura 18 – Representação de uma célula eletroquímica de dois eletrodos.	54
Figura 19 – Potencióstato para condicionamento de sensores eletroquímicos.	56
Figura 20 – Diagrama de blocos dos sistemas fixo (a) e móvel (b)	57
Figura 21 – Sensores dos fabricantes a) SPEC e b) Alphasense	59
Figura 22 – Interface entre os sensores e o microcontrolador Arduino. a) Alphasense, b) SPEC	63
Figura 23 – Interface entre o módulo cartão micro SD e o microcontrolador	64
Figura 24 – Interface entre o microcontrolador e os módulos a) RTC e b) GPS . .	64
Figura 25 – Interface entre o microcontrolador e o módulo de comunicação Wi-Fi .	65
Figura 26 – Instalação em campo do protótipo fixo	66
Figura 27 – Vista interior do protótipo fixo	67
Figura 28 – Diagrama de conexões do conjunto de sensores Alphasense	67
Figura 29 – Diagrama de conexões do conjunto de sensores Alphasense	69
Figura 30 – Sensor de Monôxido de Nitrogênio Alphasense da série B4	72
Figura 31 – Fluxograma do firmware programado para o microcontrolador Arduino MEGA	76
Figura 32 – Módulos e interfaces usados para controle e interface do RTC	85

Figura 33 – Fluxograma do firmware programado para o microcontrolador ESP8266	91
Figura 34 – Processo de atendimento a uma solicitação do mestre	95
Figura 35 – Fluxograma do processo após uma solicitação de DATA do mestre. . .	96
Figura 36 – Fluxograma do processo após uma solicitação TIME do mestre . . .	98

LISTA DE QUADROS

Quadro 1 – Modelo A.	53
----------------------	----

LISTA DE TABELAS

Tabela 1 – Requerimentos de desempenho dos instrumentos de monitoramento da qualidade do ar segundo área de aplicação	19
Tabela 2 – Especificações técnicas dos ventiladores utilizados no equipamento fixo e móvel	58
Tabela 3 – Especificações técnicas dos sensores SPEC	60
Tabela 4 – Especificações técnicas dos sensores Alphasense	61
Tabela 5 – Lista de sensores utilizados no protótipo fixo	68
Tabela 6 – Principais componentes utilizados nos dispositivos CLEAN	71
Tabela 7 – Constantes e variáveis utilizadas para controlar a execução de cada funcionalidade no firmware	83
Tabela 8 – Tipos de solicitações representadas no tipo <code>CommandTypes</code>	95

SUMÁRIO

1	INTRODUÇÃO	16
1.1	RECOMENDAÇÕES DE USO	16
1.2	OBJETIVOS	16
1.2.1	Objetivo Geral	16
1.2.2	Objetivos Específicos	16
2	MONITORAMENTO DA QUALIDADE DO AR	17
2.1	IMPORTÂNCIA	19
2.2	MONITORAMENTO DE REFERÊNCIA	19
2.3	MONITORAMENTO DE BAIXO CUSTO DA QUALIDADE DO AR	19
2.3.1	Limitações do monitoramento de baixo custo da qualidade do ar	21
3	CLEAN - COLLABORATIVE LOW-COST ENVIRONMENTAL AND AIR-QUALITY NETWORK	25
3.1	BIBLIOTECAS DE FIRMWARE	25
3.1.1	O módulo de interfaces de <i>hardware</i>	26
3.1.2	O módulo <i>drivers</i>	28
3.1.2.1	TimeDriver	28
3.1.2.2	GPSDriver	29
3.1.2.3	RTCDriver	30
3.1.2.4	DataContainer	30
3.1.2.5	HardStorage	30
3.1.3	O módulo Sensores	31
3.1.3.1	AlphaSenseISB	32
3.1.3.2	AlphaSenseCompensator	32
3.1.3.3	AlphaOXCompensator	33
3.1.3.4	Interface com sensores seriais	33
3.1.4	O módulo Data	34
3.2	A API RENOVAR	36
3.2.1	Banco de dados	38
3.2.2	A aplicação <i>Back-end</i>	39
3.2.3	A aplicação <i>Front-end</i>	39
3.3	DISPOSITIVOS DE HARDWARE DESENVOLVIDOS	40
3.3.1	Protótipos de monitores de qualidade do ar de baixo custo	41
3.3.2	A placa CLEAN Arduino MEGA	42
4	IMPLANTAÇÃO DE DISPOSITIVO EM CAMPO E RESULTADOS	45

4.1	DESCRÍÇÃO DO LOCAL (PRINCIPAIS POLUENTES, INSTRUMENTOS DE MEDAÇÃO)	45
4.2	DESCRÍÇÃO DO EXPERIMENTO (DURAÇÃO, LOCALIZAÇÃO, AMOSTRAGEM)	45
4.3	DISCUSSÃO DOS RESULTADOS	45
5	CONCLUSÃO	46
	REFERÊNCIAS	47
	APÊNDICE A – DESCRIÇÃO	53
	APÊNDICE B – SENsoRES DE GASES ELETROQUÍMICOS	54
	APÊNDICE C – PROTÓTIPOS DE MONITORES DA QUALIDADE DO AR DESENVOLVIDOS	57
C.1	TRANSPORTE DE GASES	59
C.2	SENSORIAMENTO	59
C.2.1	Sensores SPEC	60
C.2.2	Sensores Alphasense	60
C.3	CONDICIONAMENTO	61
C.3.1	Interface de condicionamento dos sensores Alphasense: A Placa de Sensoriamento Individual (ISB)	61
C.3.2	Interface de condicionamento dos sensores SPEC	62
C.4	MICROCONTROLADOR	63
C.4.1	Armazenamento dos dados	63
C.4.2	Controle de data e hora e geolocalização	63
C.4.3	Comunicação Wi-Fi	65
C.5	MONTAGEM DO PROTÓTIPO FIXO	66
C.6	MONTAGEM DO PROTÓTIPO MÓVEL	69
	APÊNDICE D – A PLACA <i>CLEAN ARDUINO MEGA</i>	70
D.1	MÓDULO DE SENSORIAMENTO	70
D.1.1	Sensores	70
D.1.1.1	Sensores SPEC.	70
D.1.1.2	Sensores Alphasense.	70
D.1.1.3	Interface de condicionamento de sensores SPEC	72
D.1.1.4	Interface de condicionamento de sensores Alphasense	72
D.2	O MICROCONTROLADOR	73
D.2.1	Armazenamento dos dados	73
D.2.2	Relógio de tempo real	73
D.2.3	Comunicação Wi-Fi	73
	APÊNDICE E – O <i>FIRMWARE CLEAN</i>	75
E.1	CÓDIGO CLEAN ARDUINO MEGA	75
E.1.1	Identificação do dispositivo e seus sensores	77

E.1.2	Configuração: a função <code>setup()</code>	77
E.1.2.1	Serial, Serial1, Serial3	79
E.1.2.2	espIoT	79
E.1.2.3	SD	79
E.1.2.4	Rtc	79
E.1.3	Interrupção Serial3	80
E.1.4	Laço principal do programa: a função <code>loop()</code>	81
E.1.4.1	Leitura dos sensores	84
E.1.4.2	Armazenamento dos dados	87
E.1.4.3	Envio de dados via protocolo <i>HTTP</i>	88
E.1.4.4	Geolocalização	88
APÊNDICE F – O FIRMWARE DO MICROCONTROLADOR		
ESP8266		90
F.1	CONFIGURAÇÃO E CONEXÃO <i>WI-FI</i>	90
F.1.1	NUM_WIFIS	91
F.1.2	WiFiCredentials wifiCreds[]	91
F.1.3	setup_wifi_connection<NUM_WIFIS>(wifiCreds)	92
F.1.4	espHTTP	92
F.1.5	espSerial	92
F.1.6	Serial	92
F.2	O LAÇO PRINCIPAL	93
F.2.1	O comando DATA: enviando um POST HTTP para a API Renovar	96
ANEXO A – DOCUMENTAÇÃO DA API RENOVAR: <i>ENDPOINTS</i>		
E REQUISIÇÕES		99

1 INTRODUÇÃO

Paralelamente ao uso deste *template* recomenda-se que seja utilizado o **Tutorial de Trabalhos Acadêmicos** (disponível neste link <https://repositorio.ufsc.br/handle/123456789/180829>) e/ou que o discente **participe das capacitações oferecidas da Biblioteca Universitária da UFSC**.

Este *template* está configurado apenas para a impressão utilizando o anverso das folhas, caso você queira imprimir usando a frente e o verso, acrescente a opção *openright* e mude de *oneside* para *twoside* nas configurações da classe *abntex2* no início do arquivo principal *main.tex* (**abntex2classe**).

Os trabalhos de conclusão de curso (TCC) de graduação e de especialização não são entregues em formato impresso na Biblioteca Universitária. Porém, sua versão PDF pode ser disponibilizada no Repositório Institucional, consulte seu curso sobre os procedimentos adotados para a entrega.

1.1 RECOMENDAÇÕES DE USO

Este *template* foi elaborado para se compilado em L^AT_EXutilizando abnT_EX2. Todas as configurações de diferenciação gráfica nas divisões de seção e subseção seguem a norma NBR 6027/2012 automaticamente.

Uma nota de rodapé, já tem seu estilo automático com o comando \footnote¹.

1.2 OBJETIVOS

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

1.2.1 Objetivo Geral

Descrição...

1.2.2 Objetivos Específicos

Descrição...

¹ As notas de rodapé possuem fonte tamanho 10. O alinhamento das linhas da nota de rodapé deve ser abaixo da primeira letra da primeira palavra da nota de modo dar destaque ao expoente.

2 MONITORAMENTO DA QUALIDADE DO AR

O monitoramento da poluição atmosférica se faz indispensável para o estudo e a gestão efetiva da qualidade do ar. O monitoramento considerado de referência é realizado por instituições governamentais através de redes de monitoramento automáticas (FRANÇA *et al.*, 2019). Essas redes são compostas por estações de monitoramento, certificadas por órgãos competentes, as quais registram ininterruptamente, e em tempo real, as concentrações dos poluentes na atmosfera (2020). As estações podem ser fixas ou móveis e a qualidade das suas leituras é garantida mediante procedimentos padrões de calibração dos instrumentos, de coleta de dados e de pós-processamento. As informações coletadas são processadas com base nos padrões legais estabelecidos e podem ser disponibilizados na forma de boletins diários ou relatórios anuais, como um resumo das condições da poluição atmosférica dentro de determinada área (CETESB, 2020). No Brasil, para facilitar o entendimento e a divulgação da informação da qualidade do ar de curto prazo, é utilizado o Índice da Qualidade do Ar *IQAr*. Este índice é um valor adimensional que outorga uma nota de Boa até Péssima à qualidade do ar, relacionando a informação quantitativa de concentração de determinados poluentes com seus efeitos na saúde humana (FRANÇA *et al.*, 2019). Os poluentes que são regulados pela legislação brasileira, na Resolução no. 491/2018, e que portanto são medidos pelas redes de monitoramento de referência são: o material particulado ($MP_{2.5-10}$), as partículas totais em suspensão, o dióxido de enxofre (SO_2), o dióxido de nitrogênio (NO_2), o ozônio ao nível de solo (O_3), o monóxido de carbono (CO) e a fumaça.

As redes de monitoramento de referência caracterizam-se por uma elevada precisão e confiabilidade. Contudo, fatores como preço, consumo energético, complexidade de operação e dimensões acabam dificultando seu uso de forma mais extensa. A literatura reporta preços dessas tecnologias em torno de \$ 15.000 – \$ 100.000 (CONCAS; MINERAUD; LAGERSPETZ; VARJONEN; LIU *et al.*, 2021) e consumo de potência aproximadamente entre 0.2 e 1 kW (PIEDRAHITA *et al.*, 2014), sem contar os custos advindos dos procedimentos de manutenção (KUMAR *et al.*, 2015). Sendo assim, a onerosidade da solução como um todo limita o número de estações financeiramente viáveis e reduz a resolução e distribuição espacial das medições, inclusive em países e regiões com elevados índices de desenvolvimento econômico como os Estados Unidos e a Europa (2015; 2016).

No Brasil o monitoramento de referência é ainda bastante limitado. Algumas iniciativas governamentais e de órgãos de pesquisa, a nível nacional e estadual, têm sido implementadas para facilitar o acesso a dados sobre a qualidade do ar como, por exemplo: (CETESB, 2020), (IEMA, 2020) e (IEMA/ES, 2020). No entanto, ainda assim, as redes de monitoramento que têm sido instaladas cobrem escassos pontos das cidades brasileiras e seu desempenho a longo prazo muitas vezes se vê comprometido por falta de

manutenção e de pessoal qualificado (OYAMA; ZAMBONI, 2017). Segundo um estudo de 2019 realizado pelo Instituto Saúde e Sustentabilidade, das 27 unidades federativas apenas 7 monitoram a qualidade do ar e atendem o regulamento vigente. O restante das unidades não realizam o monitoramento ou o realizam de forma ineficiente (INSTITUTO SAÚDE E SUSTENTABILIDADE, 2019). O estudo também aponta que das 319 estações ativas no país, a região sudeste concentra o 93% delas. O estado de Santa Catarina, em particular, não realiza monitoramento da qualidade do ar.

A pesar da sua elevada confiabilidade, devido à sua limitada distribuição espacial, as redes de monitoramento de referência da qualidade do ar proveem apenas informações globais relativas a níveis de concentrações de fundo, o que dificulta a compilação de informação confiável e representativa de áreas específicas (KUMAR *et al.*, 2015). Esse fato constitui uma limitante para o estudo dos processos associados à poluição atmosférica. Dada a alta variabilidade espaço-temporal da concentração dos poluentes, especialmente em ambientes urbanos (MEAD *et al.*, 2013), a resolução espacial das redes de monitoramento é tão relevante quanto a confiabilidade das suas estações (JIAO *et al.*, 2016).

Diante disso, se faz necessário a busca de novas soluções que possibilitem incrementar o número de monitores viáveis nas redes de monitoramento sem afetar a qualidade das medições. Todavia, especial cuidado deve ser tomado para não comprometer a qualidade dos dados obtidos, já que, como apontado por Emily Snyder e colaboradores, ter dados pouco confiáveis é mais prejudicial do que não ter dados, pois podem conduzir a decisões desacertadas (SNYDER *et al.*, 2013).

Com o intuito de regulamentar o uso de monitores de baixo custo segundo a confiabilidade das suas leituras, a Diretiva Europeia para a Qualidade do Ar definiu o Objetivo da Qualidade dos Dados (*DQO*) como requisito necessário para o uso destes instrumentos para fins de indicação da concentração de poluentes atmosféricos (EU, 2008). O *DQO* representa o nível de incerteza aceitável nas medições destes instrumentos sendo de 50% para $MP_{2.5-10}$, 30% para O_3 e 25% para CO , NO_X e SO_2 . A Agência de Proteção Ambiental Norte-americana (*US EPA*), por outro lado, sugere a avaliação dos instrumentos de monitoramento considerando cinco áreas de aplicação (Tabela 1). Cada área tem valores de precisão, viés e completude dos dados que devem ser cumpridos para que um monitor de baixo custo possa ser utilizado em aplicações dentro dessa área (WILLIAMS *et al.*, 2014). Numa direção um pouco diferente dos dois órgãos anteriores, em um estudo mais recente conduzido por Lidia Morawska, foi proposto que, dado o amplo leque de aplicações de monitoramento, os requisitos de desempenho dos monitores de baixo custo fossem definidos de acordo com cada aplicação prescindindo assim de métricas padrões (MORAWSKA *et al.*, 2018). Esta última abordagem, contudo, exige um conhecimento profundo da aplicação e dos requerimentos de desempenho associados (MORAWSKA *et al.*, 2018).

¹ Pontos de elevada concentração de determinado poluente

Tabela 1 – Requerimentos de desempenho dos instrumentos de monitoramento da qualidade do ar segundo área de aplicação

Área de aplicação	Poluentes	Erro	Completude dos dados
Educação e Informação	Todos	< 50%	$\geq 50\%$
Identificação e Caracterização de <i>Hotspots</i> ¹	Todos	< 30%	$\geq 75\%$
Monitoramento Complementar	Todos os regulados pelo CONAMA e os VOC	< 20%	$\geq 80\%$
Exposição pessoal	Todos	< 30%	$\geq 80\%$
Monitoramento de referência	O_3 CO, SO_2 NO_2 $MP_{2.5-10}$	< 7% < 10% < 15% < 10%	$\geq 75\%$

Fonte: (WILLIAMS *et al.*, 2014)

2.1 IMPORTÂNCIA

Importância (Efeitos da poluição atmosférica e necessidade de monitoramento para controle)

2.2 MONITORAMENTO DE REFERÊNCIA

(custos, equipamentos, técnicas)

2.3 MONITORAMENTO DE BAIXO CUSTO DA QUALIDADE DO AR

Segundo Emily Snyder e colaboradores o monitoramento da qualidade do ar tem experimentado uma mudança de paradigma na forma como os dados são coletados (SNYDER *et al.*, 2013). Os avanços recentes em instrumentação eletrônica unido a necessidade de soluções alternativas que complementem as técnicas de monitoramento tradicionais, têm contribuído para um crescente interesse no desenvolvimento de sensores de qualidade do ar de baixo custo (KUMAR *et al.*, 2015; LEWIS; SCHNEIDEMESSER *et al.*, 2018). Esses novos sensores possuem características essenciais como tamanho e peso reduzidos, baixo consumo de potência, baixo custo e facilidade de uso, que os colocam em vantagem com relação aos instrumentos de referência, e que têm criado as condições para aprimorar uma série de aplicações de monitoramento e gerar novas (SNYDER *et al.*, 2013; LEWIS; SCHNEIDEMESSER *et al.*, 2018).

Este tipo de tecnologia possibilitaria a agências públicas, entidades reguladoras e de pesquisa utilizar um maior volume de sistemas de monitoramento, e assim diversificar e complementar as aplicações de monitoramento para fins de pesquisa e regulamentação e validar modelos atmosféricos (LEWIS; SCHNEIDEMESSER *et al.*, 2018). Estes sistemas

podem prover informação temporal qualitativa relevante sobre o nível de poluição atmosférica em uma determinada localidade por períodos de dias a meses (CASTELL; SCHNEIDER *et al.*, 2018), como por exemplo, os momentos do dia em que a poluição é maior ou menor (ZIMMERMAN *et al.*, 2018), ou observar a sua variação ao longo do tempo (CASTELL; DAUGE *et al.*, 2017). Igualmente, os sistemas de baixo custo têm sido utilizados para detectar de áreas com níveis de concentração elevados nas cidades (MEAD *et al.*, 2013) e elaborar mapas de poluição (HUANG *et al.*, 2019).

Para Emily Snyder e colaboradores o uso deste tipo de sensores pode levar a uma melhor proteção da saúde pública e do meio ambiente (SNYDER *et al.*, 2013). Sensores de baixo custo podem ser usados para prover informação representativa de exposição pessoal a poluentes, com elevada resolução temporal (MEAD *et al.*, 2013; JERRETT *et al.*, 2017). Igualmente, esta nova forma de medição, por ser de baixo custo e de fácil operação, pode ser instalada em comunidades (MAHAJAN *et al.*, 2020), escolas e áreas residenciais (CASTELL; SCHNEIDER *et al.*, 2018), provendo informação à população sobre a qualidade do ar que respiram, e colocando os processos de monitoramento e medição nas mãos de comunidades e indivíduos (LEWIS; SCHNEIDEMESSER *et al.*, 2018).

Segundo seu princípio de funcionamento, os sensores de baixo custo utilizados para a medição de poluentes na atmosfera podem ser classificados em: sensores de material particulado e sensores de gases (MAAG; ZHOU; THIELE, 2018). Os sensores de material particulado funcionam baseados em princípios de detecção óticos que medem o espalhamento ou a adsorção da luz pelas partículas (RAI *et al.*, 2017). Já os sensores de gases podem ser subdivididos em duas classes: os óticos e os que dependem da interação entre o material transdutor e o composto gasoso (SNYDER *et al.*, 2013).

O princípio ótico de medição consiste em expor o composto gasoso a um feixe de luz, com determinado comprimento de onda, e medir o efeito dessa interação com um detector fotossensível (RAI *et al.*, 2017). Dentro desse grupo, os sensores mais comuns são os infravermelhos não-dispersivos (*NDIR*, por suas siglas em inglês), usados para medir CO_2 e CH_4 , e os detectores de foto-ionização (*PID*, por suas siglas em inglês) que utilizam luz ultravioleta e são usados para medir compostos orgânicos voláteis (SNYDER *et al.*, 2013).

Dentre os sensores de gases que operam a partir da interação entre o material transdutor e o composto gasoso, os mais populares são: os semicondutores de óxido metálico (*MOS*, por suas siglas em inglês) e os de princípio eletroquímico (*EC*, por suas siglas em inglês). Esses são os sensores mais comumente utilizados para medir gases tóxicos como CO , NO_x , O_3 e SO_2 (LEWIS; SCHNEIDEMESSER *et al.*, 2018).

Os sensores *EC*, em comparação com os *MOS*, costumam ter um menor consumo de potência, maior seletividade, menores limites de detecção e uma relação linear com a concentração. Os sensores *MOS*, por sua parte, têm custos menores e seu condicionamento

eletrônico costuma ser mais simples (RAI *et al.*, 2017). Neste trabalho serão abordados apenas os sensores eletroquímicos por serem os mais utilizados para o monitoramento de baixo custo da qualidade do ar e pela relação linear entre suas respostas e a concentração de gás.

É importante ressaltar que o atual estado da arte dos sensores de baixo custo impossibilita que eles substituam os métodos de medição de referência. Contudo, resultados promissores têm sido encontrados em várias áreas, principalmente naquelas em que as técnicas convencionais não poderiam ser implementadas devido a limitações como a pouca portabilidade, o elevado consumo energético e o custo.

2.3.1 Limitações do monitoramento de baixo custo da qualidade do ar

Os sensores de gases de baixo custo, como qualquer sistema de medição, possuem fontes de erro internas que são inerentes ao seu próprio funcionamento (MAAG; ZHOU; THIELE, 2018). Estes erros são geralmente conhecidos e fáceis de determinar. Além desses, existem fontes de interferência externas, relacionadas às suas condições de operação, que são mais difíceis de detectar e controlar. Inclusive, sensores de um mesmo fabricante e de um mesmo lote de fabricação, podem apresentar comportamentos diferentes perante a influência de fontes externas (ALPHASENSE, 2013b; CASTELL; DAUGE *et al.*, 2017).

Os fabricantes normalmente definem um intervalo de medição onde os sensores apresentam melhor desempenho. O limite inferior desse intervalo é conhecido como limite de detecção, e todos os valores inferiores a ele são considerados como ruído (MAAG; ZHOU; THIELE, 2018). Esse parâmetro é determinado em condições de laboratório, por isso em condições de operação não controladas, o seu valor pode sofrer alterações levando a erros nas medições. Por exemplo, em ambientes com muitas fontes de ruído electromagnético, ou um circuito de alimentação de energia pouco robusto a flutuações na tensão elétrica, podem aumentar a amplitude do ruído elétrico e, com ele, o limite de detecção dos sensores, afetando a sua resolução.

Outro tipo de erro sistemático que se manifesta internamente são os erros de offset e de sensibilidade. Como estes erros são não aleatórios são relativamente fáceis de remover mediante calibrações de laboratório (SPINELLE; ALEXANDRE; GERBOLES, 2013). As derivas são outra fonte interna de erros produto de alterações na sensibilidade dos sensores devido principalmente ao seu envelhecimento, que dificultam seu uso para monitoramento a longo prazo (FENG *et al.*, 2019). As derivas podem ser eliminadas com re-calibrações periódicas. A frequência das re-calibrações depende do sensor e da concentração de poluentes a que é exposto, podendo chegar a ser quinzenal (CONCAS; MINERAUD; LAGERSPETZ; VARJONEN; LIU *et al.*, 2021).

Fatores externos ao funcionamento dos sensores também produzem interferências nas medições. A influência desses fatores, principalmente das condições ambientais, são identificados por grande parte dos autores como um dos principais desafios no tratamento

das respostas dos sensores (2013; 2016; 2017; 2017). Esses problemas são característicos de medições feitas em ambientes externos, em condições e ambientes reais, não controladas, ao contrário das medições tomadas em condições de laboratório sob as quais o desempenho dos sensores costuma ser muito melhor (CASTELL; DAUGE *et al.*, 2017).

Variações na temperatura e na umidade do ambiente afetam a sensibilidade e o valor de linha base dos sensores (POPOOLA *et al.*, 2016; PANG; SHAW; GILLOT *et al.*, 2018). Particularmente, tem sido observado que em ambientes externos, onde os níveis de concentração costumam encontrar-se na ordem dos ppb, as variações na temperatura e na umidade relativa alteram o valor de linha base em maior medida que a sensibilidade (POPOOLA *et al.*, 2016).

Os fabricantes de sensores muitas vezes disponibilizam informações sobre a relação entre as respostas dos sensores e as variáveis ambientais, junto a modelos lineares de compensação (ALPHASENSE, 2013b; SPEC SENSORS, 2016a). No entanto, essas informações são obtidas a partir de testes de laboratório que simulam condições reais (SPINELLE; ALEXANDRE; GERBOLES, 2013), sendo válidas apenas em níveis de concentração na ordem dos ppm e em condições similares às dos testes (LEWIS; SCHNEIDEMESSER *et al.*, 2018). Dessa forma, as soluções para compensar os efeitos da temperatura e a umidade relativa providas pelos fabricantes resultam insuficientes para aplicações em campo (PANG; SHAW; GILLOT *et al.*, 2018). Essas informações são especialmente limitadas para aplicações de monitoramento móvel, onde os sensores são expostos a transientes de concentração bruscos e condições ambientais variadas (DELAINE; LEBENTAL; RIVANO, 2019).

Uma solução que o fabricante Alphasense tem aplicado nos seus sensores é a incorporação de um quarto eletrodo, chamado eletrodo auxiliar, cujo sinal de saída é utilizado para compensar os efeitos de variáveis ambientais no valor de linha base do sensor (ALPHASENSE, 2019a). Este eletrodo tem uma composição similar ao eletrodo de trabalho e provê um sinal de corrente de linha base que acompanha as variações do eletrodo de trabalho decorrentes das mudanças na temperatura, a umidade relativa e a pressão, podendo ser subtraída para obter a resposta do sensor ao gás (BARON; SAFFELL, 2017). Idealmente deveria funcionar assim, contudo, na prática tem sido demonstrado que o eletrodo auxiliar não é capaz de acompanhar as variações do eletrodo de trabalho em todo o intervalo de temperaturas de operação, e que portanto, uma simples subtração é insuficiente para gerar uma resposta confiável (WEI *et al.*, 2018). Cross e colaboradores também comprovaram que as correções recomendadas pelo fabricante Alphasense utilizando o eletrodo auxiliar não produzem os níveis de acurácia requeridos nas medições em ambientes externos (CROSS *et al.*, 2017).

Tem sido reportado que variações bruscas na umidade relativa e na pressão ambiente produzem picos nas respostas dos sensores que invalidam as leituras por intervalos de tempo de até 40 minutos (ALPHASENSE, 2013b; LEWIS; SCHNEIDEMESSER *et al.*, 2018).

Igualmente ambientes com valores de umidade muito extremos ou muito poluídos podem saturar os sensores ocasionando falhas e reduzindo sua sensibilidade (ALPHASENSE, 2013b).

Outros fatores que influenciam no desempenho dos sensores são as variações nos níveis de concentração do local de instalação. Os sensores eletroquímicos, por exemplo costumam ter melhor desempenho em locais onde os níveis de poluição são elevados (CASTELL; DAUGE *et al.*, 2017), já que nestas condições a dinâmica senso-gás predomina sobre o efeito das variáveis interferentes (HAGAN *et al.*, 2018). Por exemplo, Hagan e colaboradores comprovaram que o efeito da umidade relativa poderia ser ignorado ao calibrar um sensor *EC* sensível a SO_2 em todo o intervalo de medição do sensor, contudo, para medições abaixo de 25 ppb, o efeito da umidade relativa mostrou-se significativa (HAGAN *et al.*, 2018). Nuria Castell e colaboradores obtiveram coeficientes de correlação maiores com sensores instalados em locais com trânsito intenso do que nos locais com trânsito leve (CASTELL; DAUGE *et al.*, 2017). Eles também constataram que o coeficiente de correlação dos sensores testados em locais com trânsito intenso caiu de forma considerável durante o período de férias, quando o trânsito pelo local foi reduzido (CASTELL; DAUGE *et al.*, 2017). Por esse motivo, outro dos desafios dos monitores de baixo custo é que os instrumentos mantenham um bom desempenho independentemente dos níveis de concentração encontrados no local (CONCAS; MINERAUD; LAGERSPETZ; VARJONEN; PUOLAMÄKI *et al.*, 2019).

Outro problema comum dos sensores de baixo custo é a sensibilidade cruzada, que é a sensibilidade que os sensores têm a outros gases além do gás de interesse (MAAG; ZHOU; THIELE, 2018). Por exemplo, é bem conhecido que os sensores de O_3 são também sensíveis ao NO_2 (PANG; SHAW; LEWIS *et al.*, 2017; ALPHASENSE, 2019a). Outros estudos têm encontrado também sensibilidade a NO_2 em sensores de NO e SO_2 (LEWIS; LEE *et al.*, 2016), assim como a O_3 e CO_2 em sensores de NO_2 (LEWIS; SCHNEIDEMESSER *et al.*, 2018). Este efeito é especialmente desvantajoso em ambientes externos onde o ar é formado por uma mistura complexa de compostos gasosos. Por isso, se faz necessário assegurar que a leitura de um sensor corresponda ao gás para o qual foi projetado sem a interferência de outros compostos.

Uma forma de abordar o problema da sensibilidade cruzada é otimizando o material do eletrodo de trabalho durante a fabricação do sensor, para facilitar ou catalisar apenas as reações do gás de interesse (R. STETTER; LI, 2008). Igualmente, a seletividade pode ser melhorada no circuito de condicionamento, fixando o potencial de trabalho em um valor que favoreça as reações para o gás objeto de estudo (R. STETTER; LI, 2008; ALPHASENSE, 2013a). Em ambientes externos essas abordagens costumam ser insuficientes, sendo necessário utilizar arranjos de sensores nas medições e aplicar técnicas de calibração multivariadas que considerem a resposta global do arranjo (MAAG; ZHOU; THIELE, 2018).

Dada a multiplicidade e complexidade dos fatores que influenciam as medições dos sensores de baixo custo, se faz necessário o estudo cuidadoso e a aplicação de técnicas de calibração que garantam níveis de precisão e confiabilidade aceitáveis para cada aplicação de monitoramento. Isso é de importância especialmente nas aplicações de monitoramento móvel, em que os sensores são expostos a transientes bruscos nas condições de operação.

3 CLEAN - COLLABORATIVE LOW-COST ENVIRONMENTAL AND AIR-QUALITY NETWORK

A iniciativa CLEAN consiste numa plataforma colaborativa para promover e facilitar o desenvolvimento de monitores de qualidade do ar de baixo custo e o acesso remoto a informações sobre a qualidade do ar em tempo real. Possui quatro componentes principais: i) dispositivos de hardware ii) firmware reutilizável e bibliotecas de Programação Orientada a Objetos baseadas no framework Arduino para programação dos dispositivos de monitoramento, iii) guias e documentação para reprodução do hardware e adesão à rede, iv) e a plataforma Web e API Renovar para visualização, registro e acesso remoto de dados em tempo real. Todos os guias e documentação relativos ao desenvolvimento do hardware e firmware dos dispositivos até agora concebidos, as bibliotecas implementadas e as ferramentas de desenvolvimento estão abertas e disponíveis gratuitamente na página inicial de CLEAN (CAMPO, 2021).

CLEAN permite a colaboração de outros grupos e indivíduos interessados no desenvolvimento de dispositivos de monitoramento de baixo custo e dados abertos para análises ambientais. A plataforma web Renovar fornece uma API que permite que diversos dispositivos de monitoramento de baixo custo enviem seus dados para um servidor remoto para visualização e armazenamento em tempo real e geo-localizados. Além disso, a API possibilita a integração com outras aplicações Web para visualização e análise de dados. Esses dados permanecem abertamente disponíveis para posterior processamento e análise. Dada a grande versatilidade dos sensores de baixo custo, muitas aplicações poderiam ser monitoradas a partir de diversos cenários contribuindo para uma maior disponibilidade de volumes de dados. A Figura 1 ilustra os principais componentes da iniciativa CLEAN, que serão descritos a seguir.

Possui quatro componentes principais:

1. Firmware reutilizável e bibliotecas de Programação Orientada a Objetos baseadas no framework Arduino para programação dos dispositivos de monitoramento
2. A aplicação web Renovar para visualização (e acesso) remoto dos dados em tempo real
3. Dispositivos de hardware
4. Guias e documentação para reprodução do hardware

3.1 BIBLIOTECAS DE FIRMWARE

O *firmware* dos dispositivos foi desenvolvido no *Framework Arduino*, que é uma abstração de códigos-fonte e bibliotecas comuns a diversas plataformas de *hardware*. Esta estrutura torna possível escrever programas para controlar uma ampla gama de placas

microcontroladoras de Arduino e de outros fabricantes. O *framework* fornece bibliotecas de código escritas em C/C++ para programação de microcontroladores e interação com dispositivos periféricos.

Para programar todas as funcionalidades do *firmware* CLEAN, o código foi estruturado em um conjunto de classes em C++. Esta estrutura foi concebida visando sua reutilização em outras plataformas de microcontroladores e outros componentes de *hardware* suportados no Framework Arduino (como ESP8266 da Espressif) e também para facilitar a revisão e manutenção do código. As classes desenvolvidas para o projeto estão distribuídas em quatro módulos principais, conforme mostrado na Figura 1: o módulo *Hardware Interfaces*, o módulo *System Drivers*, o módulo *Sensors* e o módulo *Data*.

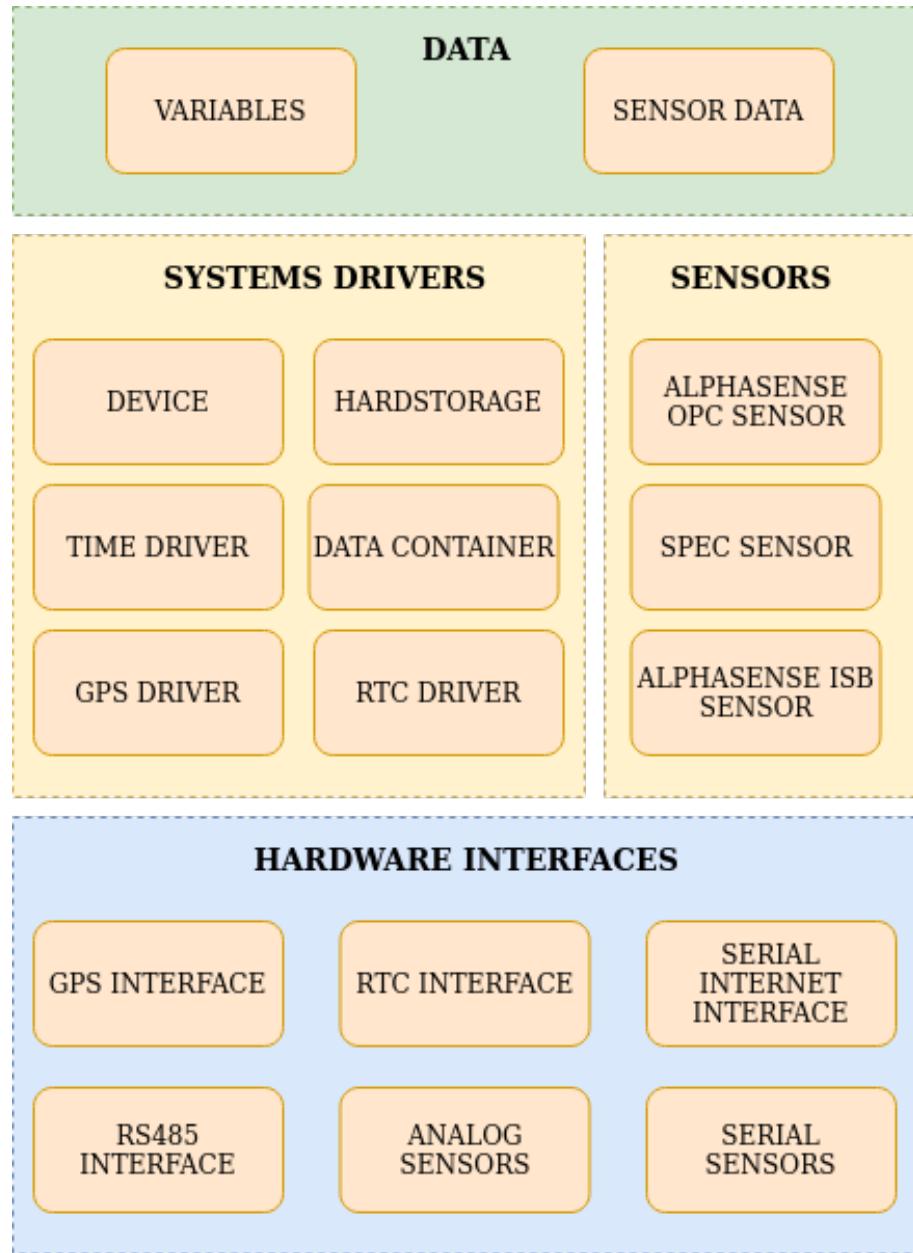
O módulo *Hardware Interfaces* agrupa todas as classes e estruturas utilizadas para se comunicar com o *hardware* periférico, como sensores, módulos de temporização, módulos de geolocalização e módulos de armazenamento. Os *Drivers* implementam funcionalidades que podem ser utilizadas pelo programa principal independentemente do *hardware* utilizado em cada dispositivo. O pacote *Sensors* está no mesmo nível dos *Drivers* e pode ser interpretado como um conjunto de *drivers* especiais para os sensores, mas com a particularidade de ser específico para cada fabricante. Por fim, o pacote Dados engloba todas as funcionalidades relacionadas à preparação de dados de sensores para armazenamento e transmissão. Este pacote abstrai as informações de concentração adquiridas pelos sensores de gás a partir de detalhes específicos sobre o funcionamento e operação de seu *hardware*.

3.1.1 O módulo de interfaces de *hardware*

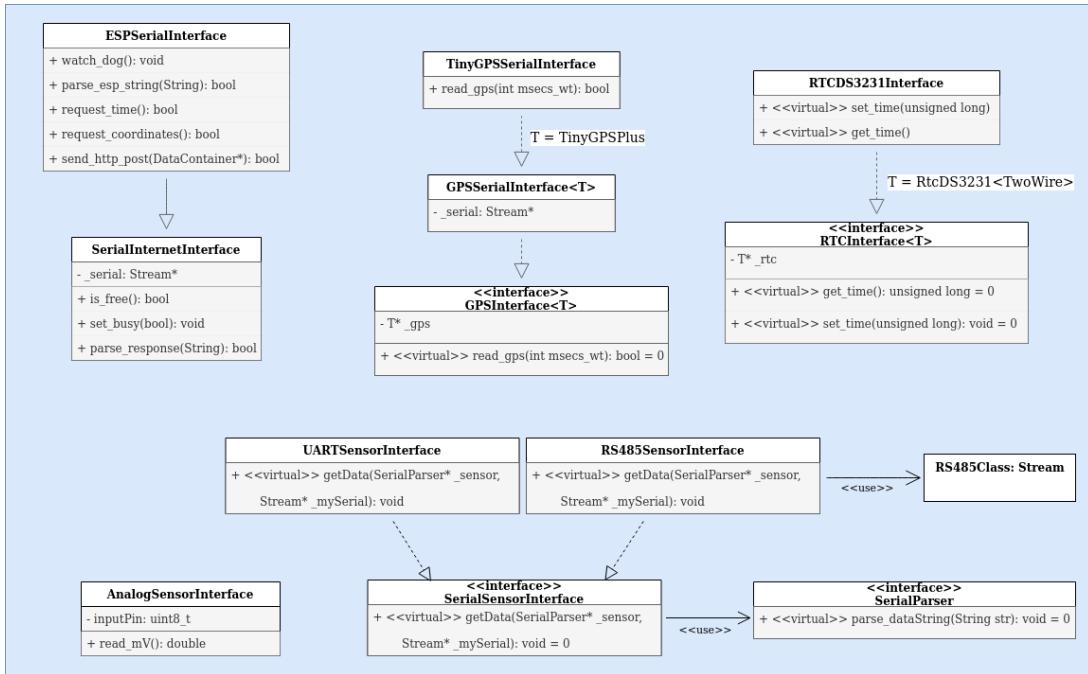
As Interfaces de *Hardware* abrangem as funcionalidades relacionadas à comunicação e interface de sensores de gás, módulos de geoposicionamento (GPS) e relógio de tempo real (RTC) que foram utilizados nos equipamentos desenvolvidos. O modo de operação e a saída dos sensores e de cada dispositivo de *hardware* determinarão o seu esquema de conexão ao microcontrolador e a forma como sua leitura é implementada no *firmware*.

A Figura 2 mostra um diagrama das classes que foram implementadas para a versão atual do *firmware*. As classes `SerialSensorInterface` e `AnalogSensorInterface` implementam interfaces para sensores digitais e analógicos, respectivamente. `SerialSensorInterface`, em particular, implementa uma interface para um sensor digital conectado através de um barramento UART ou RS-485, por meio das classes filhas `UARTSensorInterface` e `RS485SensorInterface`.

Cada classe implementa seu próprio método `sense()` que recebe como parâmetros um ponteiro para um objeto `Stream` (geralmente uma porta serial do microcontrolador), e um ponteiro para um `SerialParser`, que analisa as cadeias de caracteres com comandos ou dados enviados pelo sensor digital. O `SerialParser` é implementado numa camada superior pelas classes do módulo `Sensor`.

Figura 1 – Conjunto de bibliotecas utilizadas para *firmware* dos dispositivos CLEAN

A interface com um dispositivo serial para conexão à internet foi implementada através das classes `SerialInternetInterface` e `ESPSerialInterface`, esta última representando a conexão com o microcontrolador ESP8266. Foram criadas mais duas interfaces para módulos GPS e RTC. Na versão atual do `firmware`, foram utilizadas as bibliotecas `TinyGPSPlus` e `RtcDS3221` para cada módulo respectivamente, porém, qualquer outra biblioteca ou módulo também pode ser usado, desde que seja criado como uma classe filha de `GPSSerialInterface` e `RTCInterface`. Para isso, as classes filhas deverão implementar os métodos virtuais: `readGPS()`, `set_time()` e `get_time()` respectivamente.

Figura 2 – Diagramas de classes do pacote *Hardware Interfaces*

3.1.2 O módulo *drivers*

Os *Drivers* atuam como uma camada intermediária entre as Interfaces de *Hardware* e o programa principal. Eles abstraem o *hardware* dos dispositivos do código principal, permitindo sua reutilização independentemente dos módulos e bibliotecas utilizadas em um nível inferior. Alguns *drivers* implementados para o *firmware* foram:

- O driver **HardStorage**, para armazenamento de dados em cartão *SD*;
- O **RTCDriver** para a Interface RTC;
- O **GPSDriver** para a Interface GPS;
- O **TimeDriver** para gerenciamento das fontes de tempo no dispositivo, as quais podem provir de um módulo RTC, um módulo GPS ou de um servidor NTP

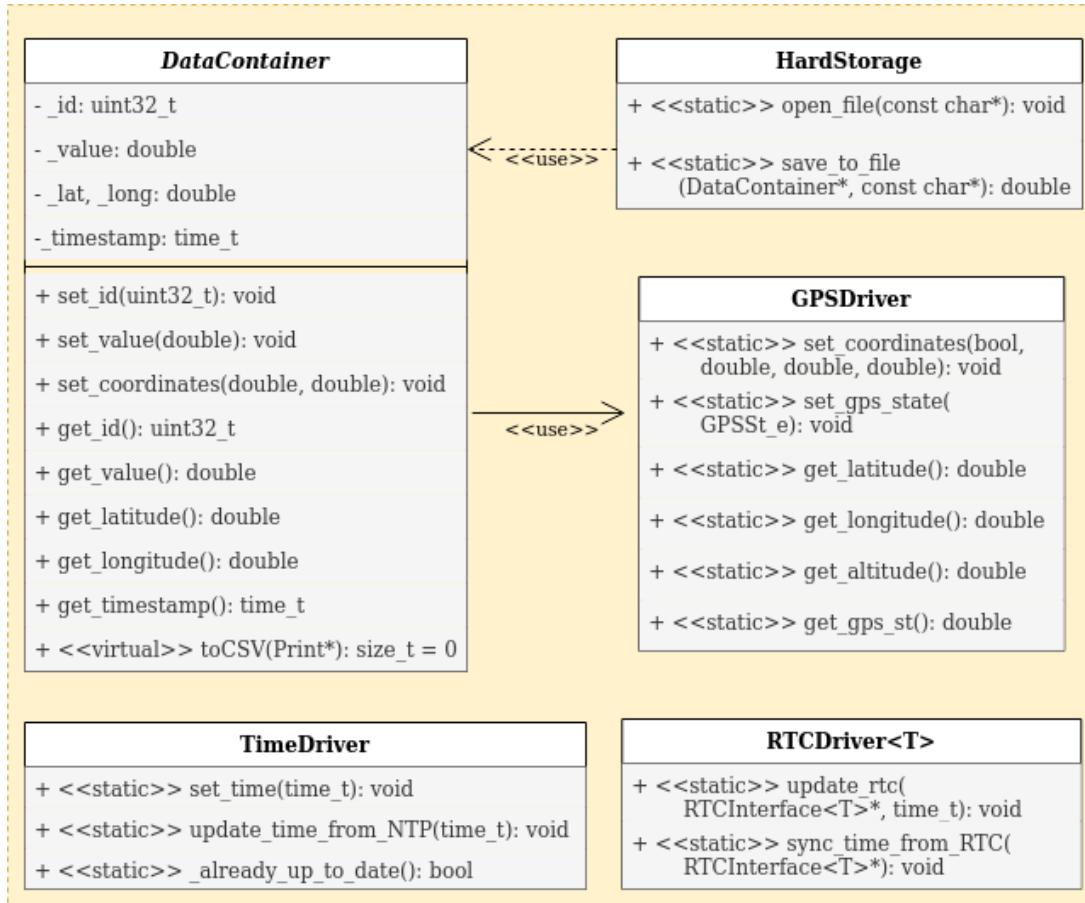
Esses quatro *drivers* usam métodos estáticos, o que significa que podem ser usados sem necessidade de ter um objeto implementado no código.

Os outros dois *drivers* que foram implementados estão relacionados ao tratamento dos dados. São eles o **DataContainer** e o **Smoother**. A Figura 3 mostra o diagrama de classes deste módulo. A continuação são resumidos alguns dos principais métodos e atributos de cada uma das classes pertencentes ao módulo **Drivers**.

3.1.2.1 TimeDriver

Esta classe registra a data e hora internas do sistema e fornece métodos para retornar informações de data e hora em diferentes formatos. O método **set_time(time_t)**

Figura 3 – Diagrama de classes do Módulo Drivers



define a data e hora do sistema. Internamente, ele invoca o método `setTime()` da biblioteca `Time.h` do framework Arduino. Recebe como parâmetro um número inteiro de 32 bits contendo a data e hora fornecidas por alguma fonte de relógio externa (um módulo GPS, um módulo `rtc` ou um servidor `ntp`).

3.1.2.2 GPSDriver

Esta classe controla a interface com um módulo `gps`, armazena as informações das coordenadas geográficas do sistema e fornece métodos para acessá-las, sendo eles:

- `static get_latitude(): double`
- `static get_longitude(): double`
- `static get_altitude(): double`
- `static get_gps_st(): GPSSt_e`

Esses métodos fornecem as informações de geolocalização armazenadas no `GPSDriver`, bem como o estado dessas informações. As informações de geolocalização

podem estar OK ou desatualizadas. Esses dois valores são retornados como uma enumeração do tipo `GPSSt_e`.

Outros dois métodos definem as coordenadas geográficas do sistema e o estado dessa informação. Esses métodos são chamados por uma instância de `GPSInterface`. Eles são:

- `static set_coordinates(): void`
- `static set_gps_state(): void`

3.1.2.3 RTCDriver

Esta classe controla a interface com um módulo RTC. O método `update_rtc(RTCInterface*, time_t)` é chamado sempre que o módulo RTC precisa ser atualizado. Recebe como parâmetro um ponteiro para a instância do `RTCInterface` que será atualizada e a data e hora. O método `sync_time_from_RTC(RTCInterface*)` retorna a data e hora a partir do ponteiro ao tipo `RTCInterface` passado como parâmetro

3.1.2.4 DataContainer

Esta é uma classe abstrata que contém informações sobre a leitura de uma variável. Essas informações são: o identificador da variável que está sendo medida e o valor dessa variável; as coordenadas; e a data e hora onde a valor foi medido. Objetos desta classe são usados para armazenar dados no cartão SD e para enviar postagens *HTTP*. O método `toCSV(Print*)` é um método virtual puro para formatar os dados de uma leitura de variável e armazená-los em um arquivo CSV. Por se tratar de um método virtual, ele deve ser implementado pelas classes filhas de `DataContainer`. Desta forma cada aplicação pode ter seu próprio formato de armazenamento das informações.

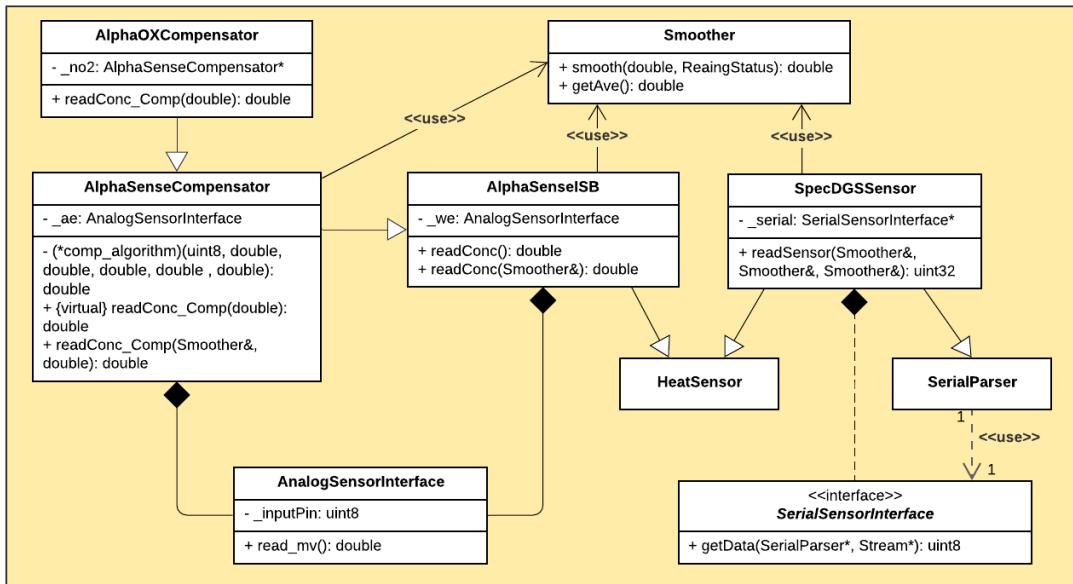
3.1.2.5 HardStorage

Esta classe contém os métodos para leitura e gravação de e para um cartão SD. Para leitura, o método `open_file(const char*)` abre o arquivo no qual as operações de leitura/gravação serão executadas. O método recebe o nome do arquivo como parâmetro. Já para a escrita no cartão, o método `save_to_file<T>(DataContainer*, const char*)` grava dados em um arquivo no cartão SD. O nome do arquivo é passado como parâmetro, juntamente com os dados a serem salvos. A função espera um ponteiro para um `DataContainer`, que na versão atual do *firmware* são objetos do tipo `SensorData`. O objeto `SensorData` implementa o método `toCSV(Print*)`, que recebe um ponteiro para o arquivo e armazena os dados nele.

3.1.3 O módulo Sensores

As classes deste pacote encapsulam a lógica de leitura de cada sensor, considerando as especificações de cada fabricante. Eles fazem uso das interfaces de sensores implementadas no pacote de Interfaces de *Hardware*. Dois fabricantes de sensores foram utilizados no *hardware* dos equipamentos desenvolvidos no contexto deste trabalho: Alphasense e SPEC Sensors. As interfaces dos sensores Alphasense e SPEC diferem na forma como foram implementadas. As saídas dos sensores Alphasense são dois sinais de tensão analógicos. Os sensores SPEC, por outro lado, fornecem os valores de concentração de gás, temperatura e umidade em uma cadeia de caracteres que é enviada através de uma interface UART. A Figura 4 mostra um diagrama das classes implementadas para este módulo.

Figura 4 – Diagramas de classes do módulo Sensors



A base para o interfaceamento dos sensores Alphasense é a leitura de duas entradas analógicas do microcontrolador utilizando a função `analogRead()` do *framework* Arduino. Por esse motivo, a classe base para modelagem dos sensores Alphasense é a classe `AnalogSensorInterface`. Ela representa uma entrada analógica identificada pelo atributo `_inputPin`, e seu método `read_mv()` converte o valor digital adquirido pelo conversor analógico-digital do Arduino, em um valor de tensão entre 0 – 5 V. Este método pode receber como parâmetro uma referência a um objeto do tipo `Smoothier`, que por sua vez deve estar associado a um objeto `Variable`. Assim, são vinculadas as variáveis físicas modeladas no *firmware* com a respectiva interface de *hardware*; neste caso uma entrada analógica.

A classe `HeatSensor` representa um sensor que precisa de um tempo de aquecimento para funcionar. A lógica que determina a validade das leituras dos sensores é implementada dentro desta classe, levando em consideração um período de aquecimento para cada sensor.

Do **HeatSensor** derivam as classes que representam os sensores Alphasense e SPEC, uma vez que ambos são sensores eletroquímicos amperométricos que requerem um intervalo de aquecimento para garantir que as leituras sejam válidas. A continuação resumem-se as principais propriedades das classes relacionadas ao interfaceamento dos sensores de gases.

3.1.3.1 AlphaSenseISB

Esta classe representa um sensor Alphasense com um circuito de condicionamento do tipo ISB. O sufixo "ISB" indica que o circuito de condicionamento usado é a placa de detecção individual do fabricante do sensor. Esta classe não incorpora nenhum algoritmo de compensação.

Atributos: `_we: AnalogSensorInterface`: Este é um atributo privado que representa a entrada analógica conectada ao eletrodo de trabalho (WE) do sensor

Métodos: `readConc(): double` `readConc(Smooth&): double` Estes são métodos públicos que convertem o valor de tensão lido pelo atributo `_we` em um valor de concentração, levando em consideração a sensibilidade do sensor informada pelo fabricante. A referência ao objeto `Smooth` associa o sensor à variável física correspondente e retorna um valor suave das leituras da variável.

3.1.3.2 AlphaSenseCompensator

Derivado do **AlphaSenseISB**, representa um sensor Alphasense com um algoritmo de compensação. Os sensores da série Alphasense B4 podem usar diferentes algoritmos de compensação dependendo do gás ao qual são sensíveis. Por isso, cada algoritmo é inerente a cada objeto e não à classe

Atributos: `_ae: AnalogSensorInterface`: Este é um atributo privado que representa a saída do eletrodo auxiliar (AE) do sensor eletroquímico. O valor de saída deste eletrodo é usado nos algoritmos de compensação.

Métodos: `(*comp_algorithm)(uint8, double, double, double, double, double): double`: Este é um ponteiro para a função que implementa o algoritmo de compensação. As funções recebem como parâmetros as variáveis necessárias para o cálculo do algoritmo, dentre eles a temperatura.

```
virtual readConc_Comp(double): double      readConc_Comp(Smooth&, double): double
```

Estes são métodos públicos que leem os valores de tensão armazenados nos atributos `_we` (herdados do **AlphaSenseISB**) e `_ae`. Eles aplicam o algoritmo de compensação correspondente e retornam um valor de concentração. Ambos os métodos recebem como parâmetros a temperatura ambiente e uma referência a um objeto `Smooth`, como no **AlphaSenseISB**.

3.1.3.3 AlphaOXCompensator

Este é um caso especial para sensores de ozônio que utilizam um algoritmo de compensação. Os sensores de ozônio medem, na verdade, a soma das concentrações de ozônio e dióxido de nitrogênio, portanto, o valor da concentração de dióxido de nitrogênio é exigido pelo algoritmo de compensação.

Atributos: `_no2: AlphaSenseCompensator*`: Para acessar o sensor de dióxido de nitrogênio, a classe `AlphaOXCompensator` usa um ponteiro para um objeto `AlphaSenseCompensator` que representa o sensor de dióxido de nitrogênio.

Métodos: `readConc_Comp(double): double`: Este método lê o valor da concentração do sensor de ozônio e aplica um algoritmo de compensação considerando também a concentração de dióxido de nitrogênio. Para vincular essas leituras a um objeto do tipo `Variable`, a classe `AlphaOXCompensator` utiliza o mesmo método `readConc_Comp()` herdado da classe `AlphaSenseCompensator`, que recebe uma referência a um objeto do tipo `Smoother`.

3.1.3.4 Interface com sensores seriais

A interface com os sensores SPEC é realizada através da classe abstrata `SerialSensorInterface`. Esta classe fornece métodos para a leitura dos sensores através da porta serial do microcontrolador. A comunicação entre os sensores e o Arduino pode ser implementada através de uma interface UART ou através de um barramento RS-485. Ambas as interfaces de comunicação são modeladas nas classes `UARTSensorInterface` e `RS485SensorInterface`, que derivam de `SerialSensorInterface`.

A classe `specDGS_sensor` funciona como uma camada intermediária entre a interface de *hardware* e as classes do módulo `Data`. Por representar um sensor eletroquímico que necessita de um período de aquecimento, esta classe também herda da classe `HeatSensor`. As instâncias de `specDGS_sensor` têm a finalidade de ler e analisar as cadeias de caracteres enviadas pelos sensores SPEC, com as medições de temperatura, umidade e concentração de gás. Esta classe também é responsável por validar as medições levando em consideração o tempo de aquecimento dos sensores e possíveis erros na comunicação serial. O método `readSensor()` lê os valores de concentração, temperatura e umidade e os disponibiliza aos objetos de tipo `Variable` correspondentes por meio das referências `Smoother` que recebe como parâmetros.

O atributo `_serial` da classe `specDGS_sensor` é um ponteiro para um objeto do tipo `SerialSensorInterface`, que é atribuído durante a construção de cada instância `specDGS_sensor`. O ponteiro pode ser um objeto do tipo `UARTSensorInterface` ou `RS485SensorInterface`, dependendo apenas da interface de comunicação implementada no *hardware*. Os objetos `specDGS_sensor` representam os sensores SPEC, e as instâncias derivadas da classe abstrata `SerialSensorInterface` representam a interface com esses

sensores, que no *hardware* é uma única porta serial.

3.1.4 O módulo Data

A Figura 5 mostra o diagrama de classes do módulo Data. Como já mencionado, este módulo funciona como uma camada intermediária que prepara e formata as medições obtidas no *hardware* do sensor para seu armazenamento e transmissão. É formado por duas classes principais: **Variable** e **SensorData**.

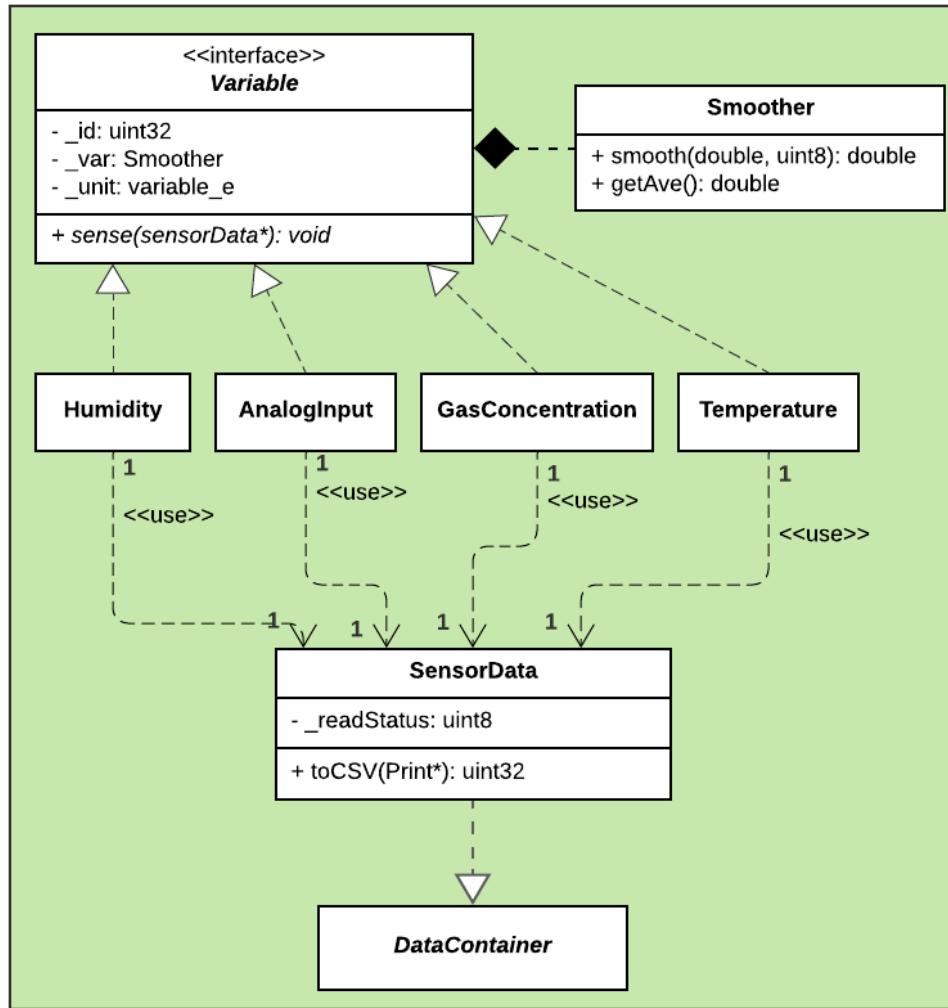
A classe **SensorData** método prepara os dados para transmissão remota e armazenamento local. Cada objeto do **SensorData** está associado a um único objeto do tipo **Variable**, que representa uma variável física com um identificador único. Vale ressaltar que, embora no *firmware* cada variável física seja representada por um único identificador, no *hardware* uma ou mais dessas variáveis podem estar vinculadas a um mesmo transdutor. O número de identificação que representa cada variável física é o que associa cada objeto da classe **Variable** ao objeto do tipo **SensorData** correspondente. Este número é armazenado em cada classe nos atributos `_sensorID` e `_id` como valores inteiros de 32 bits.

Objetos do tipo **SensorData** contêm o valor das variáveis físicas às quais estão associados, juntamente com informações sobre a data, hora e local onde as medições foram feitas. O valor de cada variável é armazenado no atributo `_value`, o qual pode ser um dado bruto medido em determinado instante de tempo ou uma média de valores adquiridos durante uma janela temporal. O método `toCSV()` consolida e prepara as informações do valor medido pelo sensor, sua geolocalização, e a data e hora em que a medição foi realizada, no formato CSV.

A classe **Variable** atua como uma camada intermediária entre a camada de *hardware* do sensor e a classe **SensorData**. Os objetos desta classe representam as variáveis físicas que estão sendo monitoradas, porém, não contêm suas quantidades, pois esses valores estão armazenados em objetos do tipo **SensorData**. Como já mencionado, o atributo `_id` contém o identificador da variável física que representa. O atributo `_unit` representa a unidade de medida da variável física que está sendo monitorada. O atributo `_var`, do tipo **Smoothen**, funciona como um buffer de memória no qual os objetos que implementam a interface de *hardware* dos sensores podem colocar as amostras da variável medida. Desse mesmo buffer, o objeto **SensorData** associado pode extrair o valor médio das amostras. O número de amostras de cada buffer depende da capacidade que for programada. Os diagramas nas Figuras 6a e 6b ilustram este processo.

Os objetos que representam os sensores, gravam as leituras de cada variável física no buffer de amostragem através do método `smooth()` da classe **Smoothen** (6a). Já a classe **Variable** acessa a média das amostras invocando o método `getAve()` do atributo `_var`. Esse valor médio é transferido para o objeto **SensorData** associado por meio do método `setValue()`. O processo de leitura e transferência do valor médio das amostras para o

Figura 5 – Diagrama de classes do pacote Data

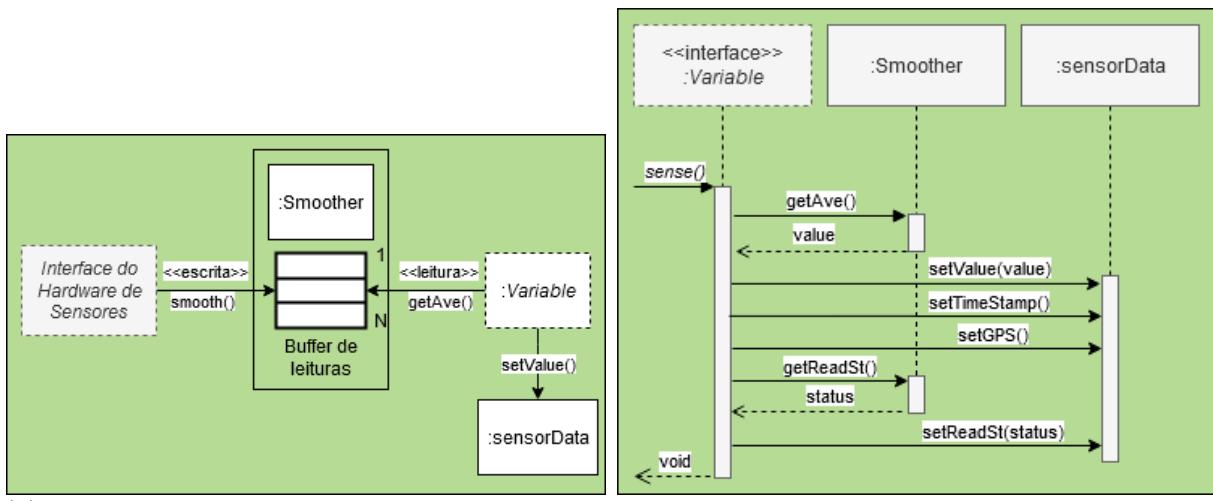


objeto **SensorData** acontece dentro do método **sense()** definido na classe **Variable**. A Figura 6b mostra o diagrama de sequência para este método.

A função **sense()** é um método virtual puro, portanto, as instâncias que derivam da classe abstrata **Variable** devem implementá-la. A sequência principal de ações executadas pelo método é comum a todas as classes derivadas. Quando o método **sense()** é invocado, a classe filha de **Variable** acessa, através do método **getAve()** de **Smoother**, o valor médio das amostras. Este valor é então passado para o objeto do tipo **SensorData** associado através do método **setValue()**. O objeto do tipo **SensorData**, por sua parte, armazena a data, a hora e o local onde foi feita a medição, bem como o status da leitura (método **getReadSt()**).

Os tipos de variáveis físicas implementadas na versão atual do *firmware* foram: Temperatura, Umidade, Concentração de gás e Entrada Analógica. Esta última representa uma tensão analógica que pode ser lida como um sinal de tensão entre 0 – 5 V. Essas variáveis foram modeladas em classes filhas de **Variable** como **Temperature**, **Humidity**, **GasConcentration** e **AnalogInput**. Por serem classes filhas, todas possuem os mesmos

Figura 6 – Processo de leitura de uma variável

(a) Processo de escrita e leitura no buffer da classe *Smoother*(b) Diagrama de sequências do método *sense()*

Fonte: Desenvolvido pelo autor (2023)

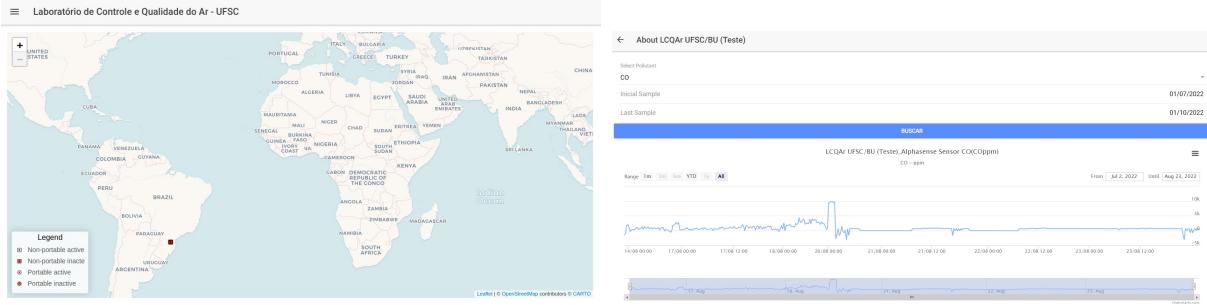
atributos de **Variable**, mas cada uma implementa seu próprio método **sense()**.

3.2 A API RENOVAR

Renovar é uma plataforma Web que fornece dados de sensores de ar para visualização e uma API para integração de sensores de ar IoT. Um MVP da plataforma foi desenvolvido em parceria com o Departamento de Informática e Estatística da UFSC (TEIXEIRA, 2018) e foi continuado no LCQAr nos anos seguintes. A plataforma é composta por um banco de dados, um serviço de *back-end* – desenvolvido em linguagem Java utilizando Spring Boot –, e uma aplicação *front-end* criada em Angular, com Ionic e TypeScript. Seu acesso é gratuito e aberto para pesquisas e análises ambientais. A Figura 7a ilustra o painel principal da plataforma, o qual consiste em um mapa que mostra a localização dos dispositivos de monitoramento. A Figura 7b ilustra o painel de séries temporais, onde o usuário consegue visualizar o histórico de determinada variável num intervalo de tempo selecionável.

O sistema recebe dados de dispositivos IoT, como concentração de poluentes atmosféricos, temperatura e umidade relativa. Os dados são armazenados em um banco de dados como séries temporais que podem ser visualizadas online na plataforma *web*. O software consiste em (1) um banco de dados MySQL que armazena as leituras do dispositivo e demais dados necessários à plataforma, como usuários, dispositivos cadastrados, poluentes e unidades; (2) um back-end RESTful, desenvolvido em Java utilizando Spring Boot, que é responsável por coletar dados do banco de dados e prepará-los para o frontend; e (3) o front-end, desenvolvido para ser multiplataforma, disponibilizando a interface com o usuário conforme ilustrado nas imagens da Figura 7.

Figura 7 – Aplicação web Renovar



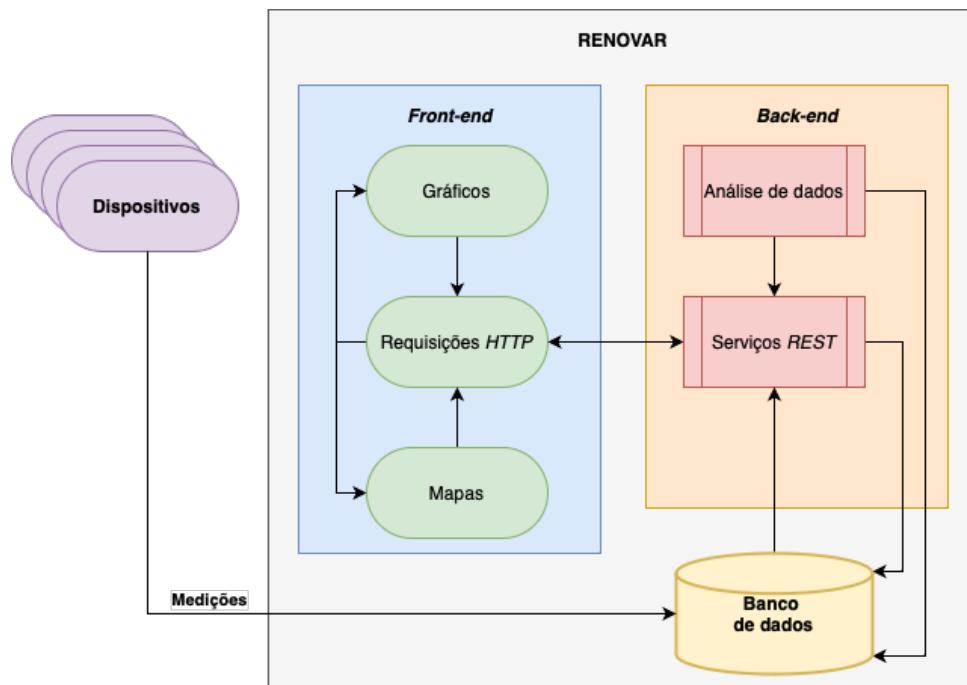
(a) Painel principal

(b) Painel de visualização de séries temporais

Fonte: Desenvolvido pelo autor (2023)

O banco de dados, backend e frontend estão hospedados em um servidor da Universidade Federal de Santa Catarina.

Figura 8 – Estrutura da aplicação Web Renovar



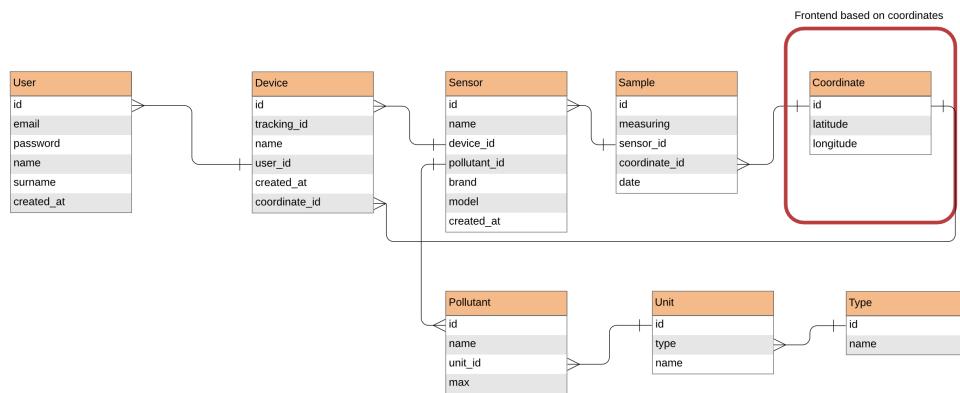
Fonte: Desenvolvido pelo autor (2023)

A Figura 8 ilustra o funcionamento do serviço em geral. Os dispositivos coletam dados ambientais e os enviam para o banco de dados pela internet. O backend recebe solicitações do frontend e coleta os dados necessários do banco de dados. Caso os dados necessitem de algum tratamento (ex.: cálculo de valores médios ou filtragem), o backend executa as operações necessárias e envia as informações processadas de volta ao frontend. O frontend, por outro lado, implementa a interface com o usuário e gera os resultados das operações solicitadas, como visualizar os dados como séries temporais e baixar os dados como arquivo CSV.

3.2.1 Banco de dados

O banco de dados foi construído utilizando *MySQL* como linguagem de consulta e *phpMyAdmin* para administração. A Figura 9 ilustra a estrutura do banco, que consiste em oito entidades, cada uma com a sua função, atributos e relacionamentos. Elas são as entidades *User*, *Device*, *Sensor*, *Sample*, *Coordinates*, *Pollutant*, *Unit* e *Type*.

Figura 9 – Entidades do banco de dados Renovar



Fonte: Desenvolvido pelo autor (2023)

A entidade *Device* representa na prática os dispositivos de coleta, mais especificamente os monitores de qualidade do ar. A entidade *Sensor* representa um sensor de determinada variável física. Relaciona-se com a entidade *Device* com uma cardinalidade 1:N, ou seja, um dispositivo pode ter N sensores enquanto um sensor pode pertencer apenas a um dispositivo. *Pollutant* representa as variáveis que são monitoradas no meio ambiente pelos dispositivos IoT. A relação entre *Pollutant* e *Sensor* também está caracterizada por uma cardinalidade 1:N, já que um sensor possui um único poluente, mas um mesmo poluente pode estar associado a N sensores. Cada poluente tem associado também uma unidade de medida e um tipo de unidade. A entidade *Sample* representa uma leitura de determinada variável física; existem N amostras por sensor. Cada amostra tem associada uma coordenada, que é a localização geográfica onde o valor da amostra será mostrado no mapa. Os dispositivos para medição em locais fixos também tem associados uma Coordenada e consequentemente, todas as amostras dos sensores desse dispositivo devem possuir os mesmos valores de latitude e longitude. Por último, a entidade *User* é utilizada para controle de acesso à plataforma. Embora o acesso aos dados de Renovar não precise de uma etapa de login, a escrita ou envio de dados para a plataforma precisa de um cadastro prévio. Assim, cada dispositivo deve ter um usuário associado para conseguir armazenar as leituras no banco de dados de Renovar.

3.2.2 A aplicação *Back-end*

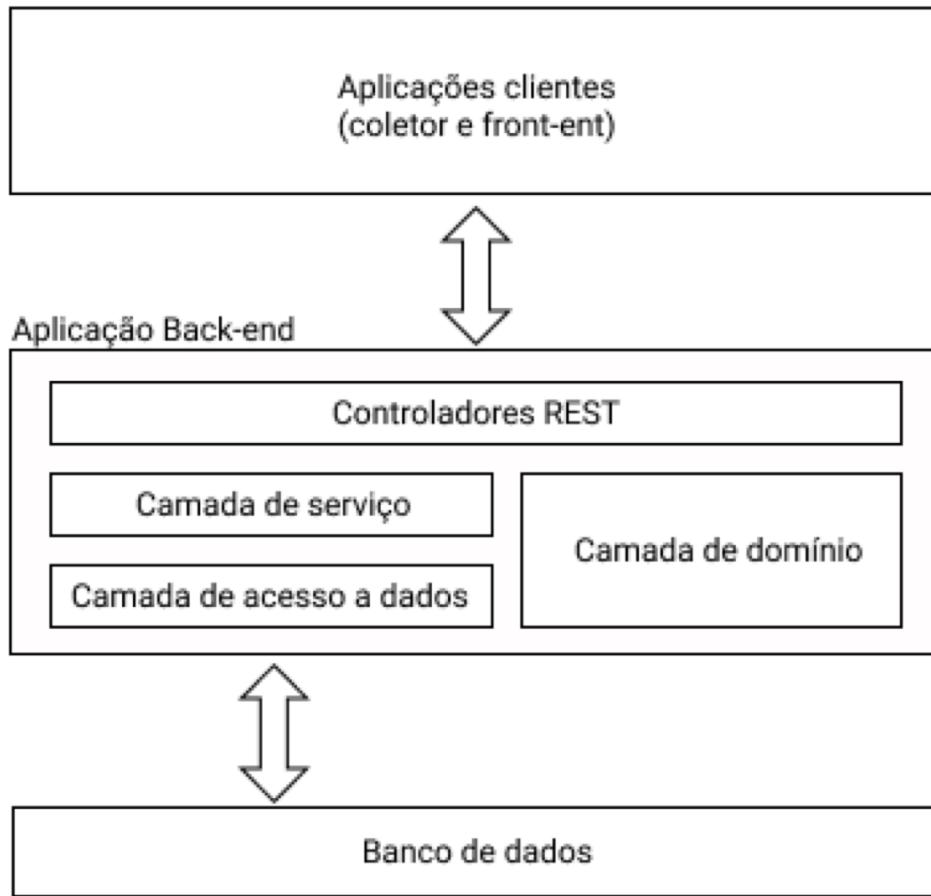
O backend é uma aplicação autônoma construída na linguagem de programação *Java*, com o *framework Spring Boot* e a ferramenta *Maven*. Esta aplicação é responsável por receber e organizar as leituras enviadas pelos dispositivos IoT, no banco de dados e atender as requisições *HTTP* do lado do cliente (*front-end*). Igualmente a aplicação realiza algumas operações básicas de análise de dados como filtragem e agrupação para gráficos tipo *box-plots*.

Quatro camadas compõem a aplicação: os controladores REST, a camada de serviço, a camada de acesso aos dados e a camada de domínio (Figura 10) (TEIXEIRA, 2018). São os controladores os que estabelecem os *endpoints*, mensagens, conteúdos e cabeçalhos de cada recurso do *back-end*. A Camada de Serviço está abaixo dos controladores REST e funciona como um mediador entre a Camada de Acesso aos Dados e a Camada de Comunicação. É ela também quem define as regras de negócio da aplicação. A Camada de Domínio replica a modelagem do banco de dados em classes que são utilizadas pelos distintos componentes da aplicação; cada classe representa uma tabela. Por último a Camada de Acesso aos Dados tem a responsabilidade de acessar diretamente o banco de dados e disponibilizar a informação para as camadas superiores. A Figura 11 ilustra as classes utilizadas na Camada de Acesso aos Dados.

O processo de coleta das medições enviadas pelos dispositivos de monitoramento na aplicação *back-end* consiste, de forma simplificada, no seguinte fluxo de execução. Os controladores REST processam as requisições de tipo *POST HTTP* enviadas pelos dispositivos IoT, e transferem as informações até a camada de serviço. Ali os dados são validados e instâncias da entidade *Sample* são criadas e transferidas até a camada de acesso aos dados onde as amostras são armazenadas na tabela correspondente do banco de dados. As requisições e os *endpoints* disponibilizados na API Renovar pelos controladores REST são detalhados na documentação da API disponibilizada no Anexo A.

3.2.3 A aplicação *Front-end*

A aplicação *front-end* foi construída usando o *framework Angular*, junto com outras ferramentas como *HighCharts* e *HighStock* para os gráficos e *Leaflet* para mapas. A tela principal mostra um mapa com todos os dispositivos (Figura 12a). A cor dos dispositivos no mapa indica seu estado: verde para ativo (i.e.: o dispositivo está transmitindo novas medidas), vermelho para inativo (o dispositivo não tem enviado novas medidas em um determinado intervalo de tempo). Quando o usuário seleciona um dispositivo no mapa, um *pop-up* é aberto com informações sobre esse e todos os dispositivos que se encontram nas mesmas coordenadas geográficas, juntamente com informações sobre seus sensores (Figura 12b). A partir desse ponto, o usuário pode acessar a página do dispositivo e escolher um poluente para visualizar sua série histórica dentro de um intervalo de datas (Figura

Figura 10 – Camadas da aplicação *back-end* Renovar

Fonte: (TEIXEIRA, 2018)

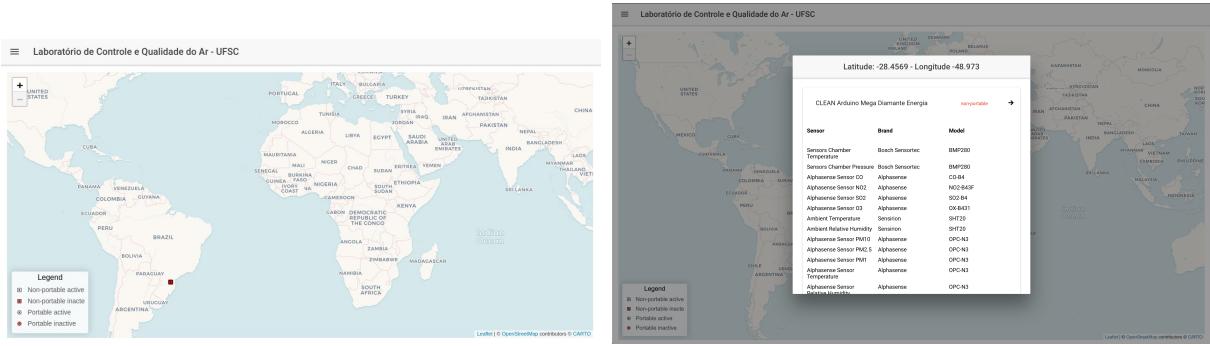
13a). Se o aparelho for portátil, o usuário também pode ver outro mapa que mostra onde cada amostra foi coletada, conforme mostrado na Figura 13b. Uma seção de análise de dados possibilita visualizar os valores das leituras dos sensores em gráficos de *box-plots*, agrupando as medições por ano mês ou semana, conforme ilustra a Figura 14.

3.3 DISPOSITIVOS DE HARDWARE DESENVOLVIDOS

Dentro do contexto da iniciativa CLEAN foram desenvolvidos dispositivos para medição da qualidade do ar. Dois desses dispositivos foram protótipos para validação da ideia, concebidos para medições em locais fixos e medições móveis. Numa segunda etapa foram desenvolvidos dispositivos mais robustos com placas de circuito impresso e quadros elétricos para instalação em campo. A continuação serão descritos os equipamentos produzidos.

Figura 11 – Classes de acesso ao dados da aplicação *back-end* Renovar

Fonte: Desenvolvido pelo autor (2023)

Figura 12 – Aplicação *front-end* da plataforma web Renovar

(a) Mapa de dispositivos

(b) Painel de seleção de dispositivos

Fonte: Desenvolvido pelo autor (2023)

3.3.1 Protótipos de monitores de qualidade do ar de baixo custo

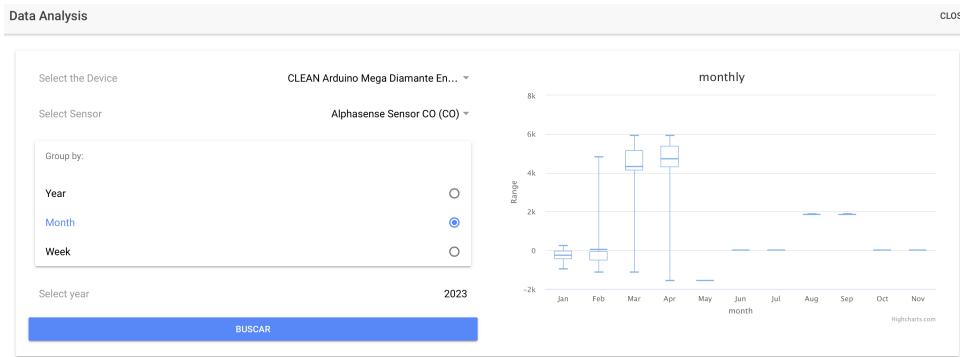
Foram concebidos dois protótipos de baixo custo para medição de poluentes atmosféricos (CAMPO *et al.*, 2020), um para monitoramento fixo e outro para monitoramento móvel. O hardware de ambos os dispositivos, conforme mostrado na Figura 15, é composto por três blocos principais: 1) transporte de gás, 2) sensoriamento e 3) microcontrolador. O estágio de transporte de gás captura o ar ambiente nos sensores, que produzem um sinal analógico proporcional à concentração do gás. O microcontrolador, que é um Microchip ATMega2560 embarcado em uma plataforma Arduino Mega, captura as respostas dos sensores e as transforma em dados de concentração de gás. O microcontrolador também obtém a hora e o local onde cada medição foi coletada. O microcontrolador armazena essas informações em um cartão micro SD e as transmite para um servidor web hospedado na Superintendência de Tecnologia da Informação e Comunicação da Universidade, rodando o aplicativo Renovar Web. Uma conexão Wi-Fi é estabelecida por um microcontrolador ESP8266 para transmissão de dados. Um relógio em tempo real e

Figura 13 – Painéis da aplicação *front-end* de Renovar

(a) Mapa de dispositivos

(b) Painel de seleção de dispositivos

Fonte: Desenvolvido pelo autor (2023)

Figura 14 – Painel de análise de dados da aplicação *back-end* Renovar

Fonte: Desenvolvido pelo autor (2023)

um módulo GPS fornecem informações de data, hora e geolocalização, respectivamente.

A versão fixa dos dispositivos de monitoramento (Figura 15a) utiliza seis sensores eletroquímicos do fabricante de sensores Alphasense e quatro sensores eletroquímicos do fabricante SPEC Sensors. Para alimentação de energia do dispositivo utiliza-se uma fonte de 12VCC. Este dispositivo não incorpora módulo *GPS* para geolocalização. A versão móvel (Figura 15b), por outro lado, utiliza apenas quatro sensores eletroquímicos do fabricante SPEC Sensors. O dispositivo é alimentado por um banco de energia de 5VCC através de uma conexão USB. A Figura 3 ilustra ambos protótipos na versão fixa e móvel. Mais detalhes sobre os dispositivos podem ser encontrados no Apêndice C.

3.3.2 A placa CLEAN Arduino MEGA

Com base nos resultados obtidos pelos protótipos e nas experiências alcançadas, foi desenvolvida uma versão mais compacta e atualizada para monitoramento fixo. Esta versão foi chamada de *CLEAN Arduino Mega Board* por causa do microcontrolador Arduino Mega que ela usa como processador principal. A composição do hardware é muito semelhante à dos protótipos, mas os módulos foram montados em uma única *PCB*. A Figura 17 ilustra o projeto da *PCB* e uma das placas fabricadas. A *PCB* foi criada no *software Eagle*, e os arquivos do projeto estão disponíveis nos repositórios do LCQAr da UFSC.

Figura 15 – Estrutura principal dos dispositivos. a) Medidor de gases fixo, e b) medidor móvel

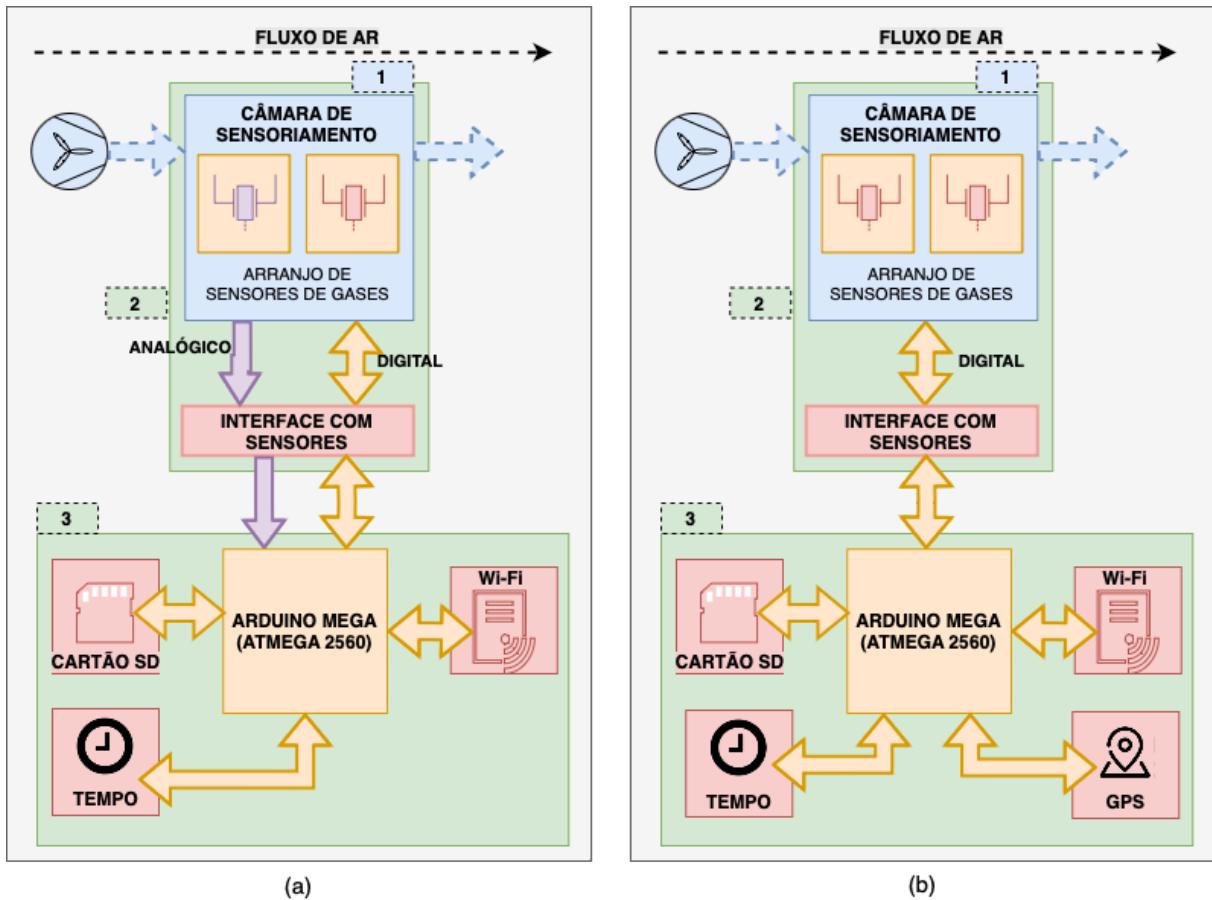
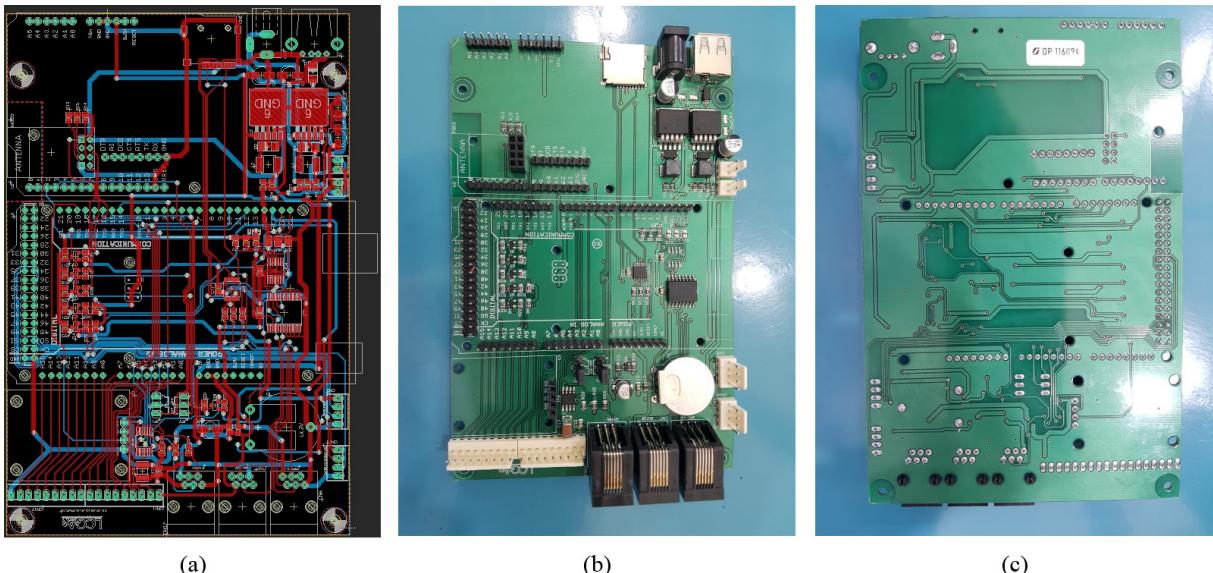


Figura 16 – Ilustrações das versões (a) fixa e (b) móvel dos dispositivos de monitoramento



Figura 17 – A placa CLEAN Arduino Mega: (a) projeto PCB, (b) vista superior da placa, (c) vista inferior da placa.



A Tabela 6 do apêndice D mostra os principais componentes de hardware utilizados na placa, que requer uma tensão de alimentação de 12V, 2A através de um conector de alimentação P4. Possui entradas analógicas para 6 placas de sensores Alphasense da série ISB, barramento RS-485 para futuras expansões, três saídas digitais e conectores para alimentação de ventoinhas de 12V e 5V. A placa foi concebida para suportar conexões Wi-Fi e *GPRS* à Internet. Essas conexões não podem ser utilizadas simultaneamente, o que dependerá de cada aplicação. O usuário pode configurar a placa para usar um ou outro e terá que adaptar o firmware do microcontrolador Arduino correspondentemente.

4 IMPLANTAÇÃO DE DISPOSITIVO EM CAMPO E RESULTADOS

- 4.1 DESCRIÇÃO DO LOCAL (PRINCIPAIS POLUENTES, INSTRUMENTOS DE MEDIÇÃO)**
- 4.2 DESCRIÇÃO DO EXPERIMENTO (DURAÇÃO, LOCALIZAÇÃO, AMOSTRAGEM)**
- 4.3 DISCUSSÃO DOS RESULTADOS**

5 CONCLUSÃO

As conclusões devem responder às questões da pesquisa, em relação aos objetivos e às hipóteses. Devem ser breves, podendo apresentar recomendações e sugestões para trabalhos futuros.

REFERÊNCIAS

- ALPHASENSE. Alphasense Application Note AAN 104 How Electrochemical Gas Sensors Work. [S.l.], 2013. P. 1–4. Disponível em:
https://www.alphasense.com/WEB1213/wp-content/uploads/2013/07/AAN_104.pdf.
- _____. Alphasense Application Note AAN 110 ENVIRONMENTAL CHANGES: TEMPERATURE, PRESSURE, HUMIDITY. [S.l.], 2013. P. 1–6. Disponível em:
https://www.alphasense.com/WEB1213/wp-content/uploads/2013/07/AAN_110.pdf.
- _____. Alphasense Application Note AAN 803-05 Correcting for Background Currents in Four Electrode Toxic Gas Sensors. [S.l.], mar. 2019. P. 1–6.
- _____. NO-B4 Nitric Oxide Sensor. Great Notley, UK, jul. 2019. P. 1–2. Disponível em: www.alphasense.com.
- BARON, Ronan; SAFFELL, John. Amperometric Gas Sensors as a Low Cost Emerging Technology Platform for Air Quality Monitoring Applications: A Review. **ACS Sensors**, v. 2, n. 11, p. 1553–1566, nov. 2017. ISSN 2379-3694. DOI: 10.1021/acssensors.7b00620. Disponível em:
<https://pubs.acs.org/doi/10.1021/acssensors.7b00620>.
- BRASIL. MINISTÉRIO DO MEIO AMBIENTE (MMA). CONSELHO NACIONAL DO MEIO AMBIENTE (CONAMA). Resolução n. 491, de 19 de novembro de 2018. Dispõe sobre padrões de qualidade do ar. [S.l.: s.n.], 2018.
- CAMPO, Fernando. The CLEAN Initiative. [S.l.: s.n.], 2021. Disponível em:
<https://lcqar.ufsc.br/novo/index.php/documentacao-clean/>.
- CAMPO, Fernando *et al.* DEPLOYMENT OF MOBILE AND FIXED AIR SENSOR PLATFORMS IN THE CITY OF FLORIANÓPOLIS, BRAZIL: PRELIMINARY RESULTS. In: 19TH Annual CMAS Conference, Chapel Hill, North Carolina. Chapel Hill, North Carolina: [s.n.], out. 2020. P. 1–6. Disponível em: <https://www.cmascenter.org/conference/2020/abstracts/Campo-et-al-2020.pdf>.
- CASTELL, Nuria; DAUGE, Franck R. *et al.* Can commercial low-cost sensor platforms contribute to air quality monitoring and exposure estimates? **Environment International**, Pergamon, v. 99, p. 293–302, fev. 2017. ISSN 0160-4120. DOI: 10.1016/J.ENVINT.2016.12.007. Disponível em:
<https://www.sciencedirect.com/science/article/pii/S0160412016309989>.
- CASTELL, Nuria; SCHNEIDER, Philipp *et al.* Localized real-time information on outdoor air quality at kindergartens in Oslo, Norway using low-cost sensor nodes.

Environmental Research, Academic Press Inc., v. 165, p. 410–419, ago. 2018. ISSN 10960953. DOI: 10.1016/j.envres.2017.10.019.

CETESB. **Redes de Monitoramento | Qualidade do Ar.** [S.l.: s.n.], 2020. Disponível em: <https://cetesb.sp.gov.br/ar/redes-de-monitoramento/>.

CONCAS, Francesco; MINERAUD, Julien; LAGERSPETZ, Eemil; VARJONEN, S.; PUOLAMÄKI, K. *et al.* A Gap Analysis of Low-Cost Outdoor Air Quality Sensor In-Field Calibration. **undefined**, 2019.

CONCAS, Francesco; MINERAUD, Julien; LAGERSPETZ, Eemil; VARJONEN, Samu; LIU, Xiaoli *et al.* **LOW-COST OUTDOOR AIR QUALITY MONITORING AND SENSOR CALIBRATION: A SURVEY AND CRITICAL ANALYSIS A PREPRINT.** [S.l.], 2021. Disponível em:
http://www.chinadaily.com.cn/china/2016-02/22/content_23595631.htm.

CROSS, Eben S. *et al.* Use of electrochemical sensors for measurement of air pollution: correcting interference response and validating measurements. **Atmospheric Measurement Techniques**, v. 10, n. 9, p. 3575–3588, set. 2017. ISSN 1867-8548. DOI: 10.5194/amt-10-3575-2017. Disponível em:
<https://www.atmos-meas-tech.net/10/3575/2017/>.

DELAINE, Florentin; LEBENTAL, Bérengère; RIVANO, Hervé. **In Situ Calibration Algorithms for Environmental Sensor Networks: A Review.** v. 19. [S.l.]: Institute of Electrical e Electronics Engineers Inc., ago. 2019. P. 5968–5978. DOI: 10.1109/JSEN.2019.2910317.

EU. **Directive 2008/50/EC of the European Parliament and of the Council of 21 May 2008 on ambient air quality and cleaner air for Europe.** [S.l.]: Official Journal of the European Union L 152/1 of 11.6.2008, 2008. P. 1–44. Disponível em:
<https://eur-lex.europa.eu/>.

FENG, Shaobin *et al.* Review on Smart Gas Sensing Technology. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 19, n. 17, p. 3760, ago. 2019. ISSN 1424-8220. DOI: 10.3390/s19173760. Disponível em:
<https://www.mdpi.com/1424-8220/19/17/3760>.

FRANÇA, André Luiz Felisberto *et al.* **GUIA TÉCNICO PARA O MONITORAMENTO E AVALIAÇÃO DA QUALIDADE DO AR.** Brasília, DF, 2019. P. 1–135. Disponível em:
<https://www.mma.gov.br/agenda-ambiental-urbana/qualidade-do-ar.html>.

HAGAN, David H. *et al.* Calibration and assessment of electrochemical air quality sensors by co-location with regulatory-grade instruments. **Atmospheric Measurement Techniques**, Copernicus GmbH, v. 11, n. 1, p. 315–328, jan. 2018. ISSN 18678548. DOI: 10.5194/amt-11-315-2018.

HUANG, Keyong *et al.* Estimating daily PM2.5 concentrations in New York City at the neighborhood-scale: Implications for integrating non-regulatory measurements. **Science of the Total Environment**, Elsevier B.V., v. 697, p. 134094, dez. 2019. ISSN 18791026. DOI: 10.1016/j.scitotenv.2019.134094.

IEMA. **Qualidade do Ar.** [S.l.: s.n.], 2020. Disponível em: <http://www.qualidadedoar.org.br/>.

IEMA/ES. **IEMA - Qualidade do Ar.** [S.l.: s.n.], 2020. Disponível em: <https://iema.es.gov.br/qualidadedoar/redesdemonitoramento/automaticaramqar>.

INSTITUTO SAÚDE E SUSTENTABILIDADE. **Análise do Monitoramento de Qualidade do Ar no Brasil.** [S.l.], 2019. P. 1–20.

JERRETT, Michael *et al.* Validating novel air pollution sensors to improve exposure estimates for epidemiological analyses and citizen science. **Environmental Research**, Academic Press, v. 158, p. 286–294, out. 2017. ISSN 0013-9351. DOI: 10.1016/J.ENVRES.2017.04.023. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0013935117307351?via%3Dihub>.

JIAO, Wan *et al.* Community Air Sensor Network (CAIRSENSE) project: evaluation of low-cost sensor performance in a suburban environment in the southeastern United States. **Atmospheric Measurement Techniques**, v. 9, p. 5281–5292, 2016. DOI: 10.5194/amt-9-5281-2016. Disponível em: www.atmos-meas-tech.net/9/5281/2016/.

KUMAR, Prashant *et al.* The rise of low-cost sensing for managing air pollution in cities. **Environment International**, Pergamon, v. 75, p. 199–205, fev. 2015. ISSN 0160-4120. DOI: 10.1016/J.ENVINT.2014.11.019. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0160412014003547?via%3Dihub>.

LEWIS, Alastair C.; LEE, James D. *et al.* Evaluating the performance of low cost chemical sensors for air pollution research. **Faraday Discussions**, The Royal Society of Chemistry, v. 189, n. 0, p. 85–103, jul. 2016. ISSN 1359-6640. DOI: 10.1039/C5FD00201J. Disponível em: <http://xlink.rsc.org/?DOI=C5FD00201J>.

LEWIS, Alastair C.; SCHNEIDEMESSER, Erika von *et al.* **Low-cost sensors for the measurement of atmospheric composition: overview of topic and future applications.** Geneva, mai. 2018. P. 1–60. ISBN 9789263112156. Disponível em: http://www.wmo.int/pages/prog/arep/gaw/documents/Draft_low_cost_sensors.pdf%20http://eprints.whiterose.ac.uk/135994/#.XmE_tzYtmFc.mendeley.

MAAG, Balz; ZHOU, Zimu; THIELE, Lothar. A Survey on Sensor Calibration in Air Pollution Monitoring Deployments. **IEEE Internet of Things Journal**, Institute of Electrical e Electronics Engineers Inc., v. 5, n. 6, p. 4857–4870, dez. 2018. ISSN 23274662. DOI: 10.1109/JIOT.2018.2853660.

MAHAJAN, Sachit *et al.* A citizen science approach for enhancing public understanding of air pollution. **Sustainable Cities and Society**, v. 52, p. 101800, 2020. ISSN 2210-6707. DOI: <https://doi.org/10.1016/j.scs.2019.101800>. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2210670719317020>.

MEAD, M.I. *et al.* The use of electrochemical sensors for monitoring urban air quality in low-cost, high-density networks. **Atmospheric Environment**, Pergamon, v. 70, p. 186–203, mai. 2013. ISSN 1352-2310. DOI: 10.1016/J.ATMOSENV.2012.11.060. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1352231012011284>.

MORAWSKA, Lidia *et al.* Applications of low-cost sensing technologies for air quality monitoring and exposure assessment: How far have they gone? **Environment International**, Pergamon, v. 116, p. 286–299, jul. 2018. ISSN 0160-4120. DOI: 10.1016/J.ENVINT.2018.04.018. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0160412018302460?via%3Dihub>.

OYAMA, Beatriz Sayuri; ZAMBONI, Ademilson. AIR QUALITY MONITORING AND DATA AVAILABILITY IN BRAZIL. In: _____. **Air Quality Conference Brazil, 3rd CMAS South America**. Vitória - ES, Brazil: Fundação Espírito Santo de Tecnologia, 2017. P. 170–172.

PANG, Xiaobing; SHAW, Marvin D.; GILLOT, Stefan *et al.* The impacts of water vapour and co-pollutants on the performance of electrochemical gas sensors used for air quality monitoring. **Sensors and Actuators B: Chemical**, v. 266, p. 674–684, 2018. ISSN 09254005. DOI: 10.1016/j.snb.2018.03.144.

PANG, Xiaobing; SHAW, Marvin D.; LEWIS, Alastair C. *et al.* Electrochemical ozone sensors: A miniaturised alternative for ozone measurements in laboratory experiments and air-quality monitoring. **Sensors and Actuators, B: Chemical**, Elsevier B.V., v. 240, p. 829–837, mar. 2017. ISSN 09254005. DOI: 10.1016/j.snb.2016.09.020.

PIEDRAHITA, R. *et al.* The next generation of low-cost personal air quality sensors for quantitative exposure monitoring. **Atmospheric Measurement Techniques**, v. 7, n. 10, p. 3325–3336, out. 2014. DOI: 10.5194/amt-7-3325-2014. Disponível em: <https://www.atmos-meas-tech.net/7/3325/2014/>.

POPOOLA, Olalekan A.M. *et al.* Development of a baseline-temperature correction methodology for electrochemical sensors and its implications for long-term stability. **Atmospheric Environment**, Pergamon, v. 147, p. 330–343, dez. 2016. ISSN 1352-2310. DOI: 10.1016/J.ATMOSENV.2016.10.024. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1352231016308317?via%3Dihub>.

R. STETTER, Joseph; LI, Jing. Amperometric Gas SensorsA Review. **Chemical Reviews**, v. 108, n. 2, p. 352–366, jan. 2008. DOI: 10.1021/cr0681039.

RAI, Aakash C. *et al.* **End-user perspective of low-cost sensors for outdoor air pollution monitoring.** [S.l.: s.n.], 2017. DOI: 10.1016/j.scitotenv.2017.06.266.

SNYDER, Emily G. *et al.* The Changing Paradigm of Air Pollution Monitoring. **Environmental Science & Technology**, American Chemical Society, v. 47, n. 20, p. 11369–11377, out. 2013. ISSN 0013-936X. DOI: 10.1021/es4022602. Disponível em: <http://pubs.acs.org/doi/10.1021/es4022602>.

SPEC SENSORS. **Application Note AN-104 Short-Term Environment Effects on Performance.** [S.l.], 2016. P. 1–6. Disponível em:
<http://www.spec-sensors.com/wp-content/uploads/2016/06/SPEC-AN-104-Environmental-Effects.pdf>.

_____. **Digital Gas Sensor Developer Kit 968-045.** [S.l.], 2017. P. 1–10. Disponível em: https://www.spec-sensors.com/wp-content/uploads/2017/01/DG-SDK-968-045_9-6-17.pdf.

_____. **SPEC Sensor Operation Overview SPEC Sensor TM Operation and Performance Considerations.** [S.l.], 2016. P. 1–6. Disponível em:
<http://www.spec-sensors.com/wp-content/uploads/2016/05/SPEC-Sensor-Operation-Overview.pdf>.

SPINELLE, Laurent; ALEIXANDRE, Manuel; GERBOLES, Michel. **Protocol of evaluation and calibration of low-cost gas sensors for the monitoring of air pollution.** [S.l.], 2013. P. 46. ISBN 9789279326912. DOI: 10.2788/9916. Disponível em: <http://publications.jrc.ec.europa.eu/repository/handle/JRC83791>.

TEIXEIRA, Francisco Sacco Flores Almeida. Renovar: Um MVP para monitorar a qualidade do ar, p. 1–157, 2018. Disponível em:
<https://repositorio.ufsc.br/handle/123456789/192181>.

WEI, Peng *et al.* Impact Analysis of Temperature and Humidity Conditions on Electrochemical Sensor Response in Ambient Air Quality Monitoring. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 18, n. 2, p. 59, jan. 2018. ISSN 1424-8220. DOI: 10.3390/s18020059. Disponível em:
<http://www.mdpi.com/1424-8220/18/2/59>.

WESTBROEK, P. Fundamentals of electrochemistry. In: ANALYTICAL Electrochemistry in Textiles. [S.l.]: Elsevier Inc., jan. 2005. P. 3–36. ISBN 9781855739192. DOI: 10.1533/9781845690878.1.1.

WILLIAMS, Ron *et al.* **Air Sensor Guidebook.** Washington, DC: U.S. Environmental Protection Agency, EPA/600/R-14/159 (NTIS PB2015-100610), 2014. P. i–65. Disponível em: https://cfpub.epa.gov/si/si_public_record_report.cfm?Lab=NERL&dirEntryId=277996.

ZIMMERMAN, Naomi *et al.* A machine learning calibration model using random forests to improve sensor performance for lower-cost air quality monitoring. **Atmospheric Measurement Techniques**, v. 11, n. 1, p. 291–313, jan. 2018. ISSN 1867-8548. DOI: 10.5194/amt-11-291-2018. Disponível em:
<https://www.atmos-meas-tech.net/11/291/2018/>.

APÊNDICE A – DESCRIÇÃO

Textos elaborados pelo autor, a fim de completar a sua argumentação. Deve ser precedido da palavra APÊNDICE, identificada por letras maiúsculas consecutivas, travessão e pelo respectivo título. Utilizam-se letras maiúsculas dobradas quando esgotadas as letras do alfabeto.

Quadro 1 – Modelo A.

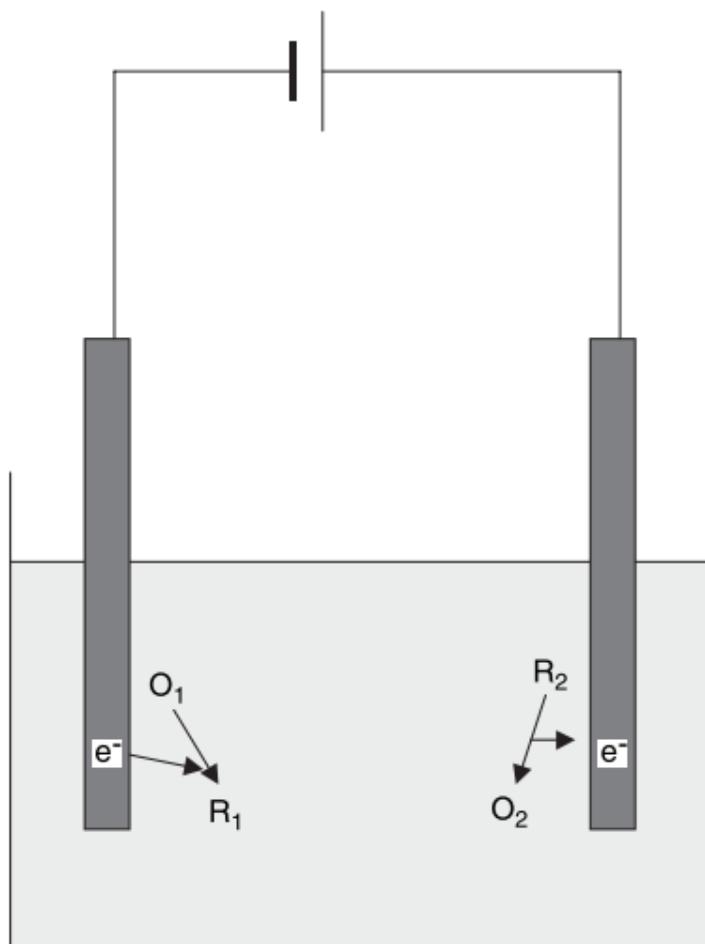
xxxx	yyyyyyyyyyyyyyyyyy
rrrrrrrrrrrrrrrrrr	eeeeeeeeeeeeeeeeee
xxxx	yyyyyyyyyyyyyyyyyy
xxxx	yyyyyyyyyyyyyyyyyy
rrrrrrrrrrrrrrrrrr	eeeeeeeeeeeeeeeeee
xxxx	yyyyyyyyyyyyyyyyyy
	ttttttttttttttt
rrrrrrrrrrrrrrrrrr	eeeeeeeeeeeeeeeeee
ttttttttttttt	
rrrrrrrrrrrrrrrrrr	eeeeeeeeeeeeeeeeee
rrrrrrrrrrrrrrrrrr	eeeeeeeeeeeeeeeeee
	gggggggggggggggggg
rrrrrrrrrrrrrrrrrr	eeeeeeeeeeeeeeeeee

Fonte: Elaborada pelo autor (2016).

APÊNDICE B – SENSORES DE GASES ELETROQUÍMICOS

Os sensores eletroquímicos funcionam baseados no princípio de conversão de energia química em energia elétrica e vice-versa, e consistem, basicamente, em uma célula contendo uma solução eletrolítica na qual são submersos eletrodos metálicos interconectados por um circuito externo (WESTBROEK, 2005). Dentro da célula originam-se reações de oxidação-redução entre cada um dos eletrodos e a solução eletrolítica. As reações de redução removem elétrons do material do eletrodo, que passa a funcionar como catodo. Nas reações de oxidação, o eletrodo que faz o papel de anodo ganha elétrons e produz-se uma espécie oxidada. Esse intercâmbio de cargas entre os eletrodos e a solução analítica, equivalente a uma corrente elétrica fluindo do anodo para o catodo, é proporcional à velocidade das reações redox nas superfícies dos eletrodos (R. STETTER; LI, 2008). A Figura 18 mostra a estrutura básica de uma célula eletrolítica amperométrica de dois eletrodos.

Figura 18 – Representação de uma célula eletroquímica de dois eletrodos.



Fonte: (WESTBROEK, 2005).

Segundo seu princípio de operação, os sensores eletrolíticos são classificados em amperométricos, potenciométricos e condutimétricos (R. STETTER; LI, 2008). Neste trabalho apenas são abordados os sensores amperométricos por serem os mais comumente utilizados no monitoramento de baixo custo.

Os sensores amperométricos apresentam uma estrutura básica de três eletrodos: o eletrodo de trabalho, o eletrodo contador e o eletrodo de referência. O eletrodo de trabalho é a superfície onde acontece a reação de interesse entre o material do eletrodo, a solução eletrolítica e o gás sob estudo, que formam uma interface de três fases. Para aumentar a seletividade dos sensores, costuma-se aplicar algum catalisador na superfície do eletrodo para facilitar ou catalisar determinadas reações. Como este eletrodo está em contato direto com o ar ambiente, corre risco de envenenamento se exposto a certos gases que possam ser adsorvidos no catalisador ou que possam reagir com ele produzindo outros compostos químicos que inibam sua ação (ALPHASENSE, 2013a; WESTBROEK, 2005; R. STETTER; LI, 2008; BARON; SAFFELL, 2017).

A reação no eletrodo de trabalho pode ser de oxidação ou de redução, dependendo da natureza da substância gasosa de interesse. Na ausência de gás, a célula eletrolítica encontra-se em equilíbrio, mas, ao entrar em contato com a substância gasosa, as moléculas do gás produzem um desbalanço nas reações redox, e, como resultado, o eletrodo pode ganhar ou perder elétrons, obtendo assim uma carga elétrica (ALPHASENSE, 2013a; R. STETTER; LI, 2008).

A carga elétrica gerada no eletrodo de trabalho é balanceada com uma reação oposta no eletrodo contador. Se a reação no eletrodo de trabalho for de oxidação, então no contador se produzirá uma reação de redução complementar, e vice-versa. Como resultado, uma corrente elétrica é gerada, proporcional à velocidade das reações nas superfícies dos eletrodos, que por sua vez é proporcional à concentração do gás (ALPHASENSE, 2013a; R. STETTER; LI, 2008; WESTBROEK, 2005).

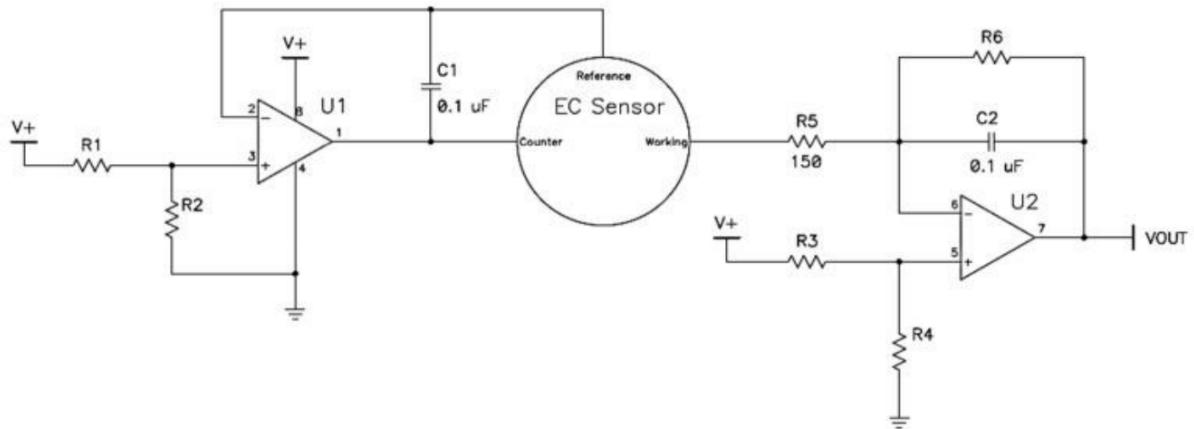
Para assegurar que o sensor se encontra operando dentro de uma região de trabalho desejada, é utilizado o eletrodo de referência. Ele é encarregado de fixar a tensão do eletrodo de trabalho em um valor constante sem que este valor (i.e.: a tensão de operação) seja alterado pela circulação da corrente elétrica, entre os eletrodos contador e de trabalho, que resulta dos processos de transdução. Já o eletrodo contador é deixado com uma carga flutuante que dependerá unicamente das reações eletroquímicas decorrentes da exposição ao gás e do fluxo de elétrons resultante. A corrente de saída do sensor é o resultado da diferença de potencial entre os eletrodos contador e de trabalho (ALPHASENSE, 2013a; BARON; SAFFELL, 2017).

A corrente de saída dos sensores eletroquímicos é muito pequena, geralmente na ordem dos nanoampères (R. STETTER; LI, 2008), fazendo necessária a utilização de uma etapa posterior de condicionamento. A função principal da etapa de condicionamento é transformar o valor da variável elétrica de saída do transdutor em um dado que possa ser

lido por um sistema de aquisição e que represente a quantidade física sendo medida, em um determinado instante de tempo. Comumente, a saída da etapa de condicionamento é um sinal de tensão, ajustado para representar, proporcionalmente, dentro de um determinado intervalo, a variável física de interesse. Esse sinal de tensão pode então ser lido por um conversor analógico-digital. Adicionalmente, podem ser contempladas subetapas intermediária de amplificação, filtragem ou alisamento, ajuste de zero e de ganho, e linearização.

No caso dos sensores eletroquímicos, a configuração mais utilizada no circuito de condicionamento é o potenciostato (ALPHASENSE, 2013a; SPEC SENSORS, 2016b). Este circuito controla o potencial aplicado aos eletrodos de trabalho e de referência e converte a corrente que circula entre os eletrodos de trabalho e contador em um valor de tensão proporcional. Um diagrama simplificado de um sensor eletroquímico e um potenciostato é ilustrado na Figura 19.

Figura 19 – Potenciostato para condicionamento de sensores eletroquímicos.



Fonte: (SPEC SENSORS, 2016b).

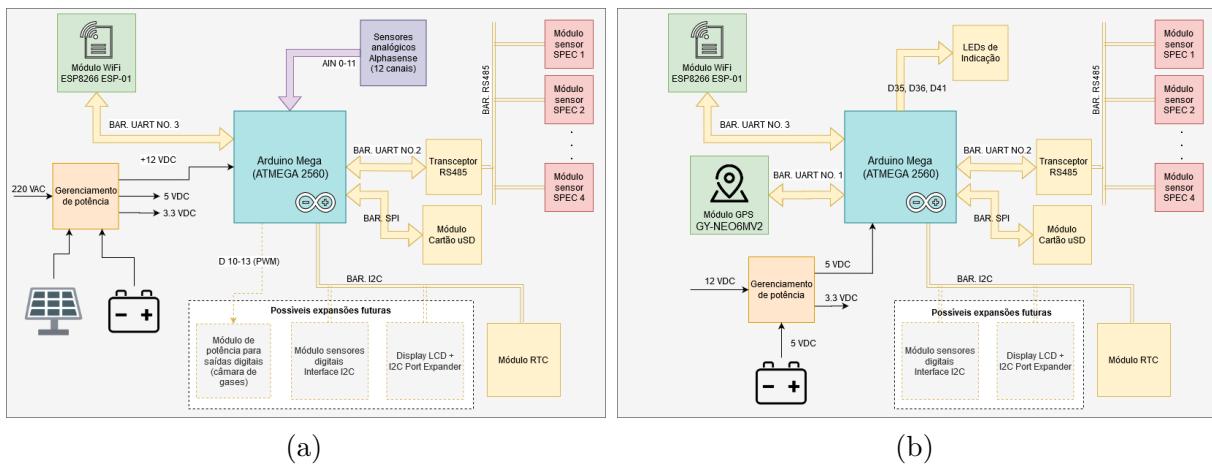
O potencial do eletrodo de referência é estabelecido pelo amplificador U1 da Figura 19, no pino 2. O amplificador U1 também garante a circulação da corrente elétrica pelo eletrodo contador, mantendo constante a tensão de referência. O amplificador U2 fixa o potencial do eletrodo de trabalho no pino 5, e converte em tensão (VOUT) a corrente que circula entre esse eletrodo e o contador. Dessa forma, é possível medir a tensão no pino VOUT como um valor proporcional à concentração do gás em contato com a superfície do sensor (SPEC SENSORS, 2016b).

APÊNDICE C – PROTÓTIPOS DE MONITORES DA QUALIDADE DO AR DESENVOLVIDOS

Foram desenvolvidos dois protótipos para o monitoramento da qualidade do ar ilustrados na Figura 16 do Capítulo 3. Eles foram baseados na plataforma Arduino Mega 2560, que utiliza o microcontrolador ATMega2560 da Microchip. Um deles foi projetado para medição fixa em um local, e o outro para monitoramento de forma móvel. Este último, além de prover a informação temporal associada a cada leitura de concentração de poluente, inclui a localização onde a medição foi tomada. Os dispositivos foram projetados para a medição de poluentes regulados na Resolução CONAMA No. 491/2018, sendo eles: CO , NO_2 , SO_2 e O_3 . Além desses gases, a versão fixa também inclui um sensor de sulfeto de hidrogênio (H_2S).

A Figura 20 mostra diagramas com os módulos de *hardware* que compõem os sistemas de medição fixo e móvel, sem incluir o processo de transporte dos gases. A versão fixa do sistema de monitoramento (Figura 20a) utiliza seis sensores da empresa Alphasense sensíveis aos gases CO , NO_2 , SO_2 , O_3 e H_2S . Além destes, também estão instalados quatro sensores da empresa SPEC, sensíveis aos mesmos poluentes com exceção do H_2S . A conexão entre a plataforma Arduino Mega e os sensores SPEC é realizada pela porta serial UART2 do microcontrolador através de um barramento RS-485. Já a leitura dos sensores da Alphasense é realizada pelas entradas analógicas AI0 – AI11 do microcontrolador.

Figura 20 – Diagrama de blocos dos sistemas fixo (a) e móvel (b)



Fonte: Desenvolvido pelo autor (2023)

A versão móvel (Figura 20b) utiliza apenas sensores da SPEC para a medição de gases. Os modelos SPEC utilizados nesta versão são os mesmos que na versão fixa, e utilizam a mesma configuração para se comunicar de forma serial com o Arduino Mega. Um diferencial desta versão com relação à fixa, além de não utilizar sensores Alphasense, é

Tabela 2 – Especificações técnicas dos ventiladores utilizados no equipamento fixo e móvel

Características	Versão fixa	Versão móvel
Descrição	Ventilador cooler 40mm 12VDC	Ventilador cooler 40mm 5VDC
Marca	GC	GDT
Tamanho	40 x A: 40 x C: 10mm	L: 40 x A: 40 x C: 10mm
Corrente nominal	80 ± 10% mA	140 ± 10% mA
Tensão nominal	12 V	Entre 5 e 7V
Ruído	16 ± 10% dBA	16 ± 10% dBA
Velocidade rotação	5000 ± 10% RPM	7000 ± 20% RPM
Fluxo de ar	4.2 CFM	6.12 CFM
Peso	12 g	14 g
Consumo potência	1.2 W	0.8 W
Vida útil	35000 hr	50000 hr

a inclusão de um módulo GPS para georreferenciar as medições dos poluentes. O módulo utilizado é o GY-NEO6MV2 que se comunica com o microcontrolador através da UART1.

Além dos dispositivos mencionados acima, cada unidade de monitoramento inclui um módulo ESP8266 ESP-01 conectado à porta serial UART3 do Arduino, para comunicação Wi-Fi. Ambas unidades utilizam também um módulo de cartão micro SD para o armazenamento dos dados e um módulo de Relógio de Tempo Real (RTC) para manter a informação de data e hora. Outros periféricos como sensor de pressão, monitor LCD ou atuadores para controlar o transporte dos gases, podem ser adicionados através do barramento I2C em versões futuras.

Os módulos que compõem ambos sistemas de medição funcionam com tensões de alimentação tanto de 3.3 V como 5 V. Para fornecer esses níveis de voltagem foi utilizado um módulo-fonte que utiliza dois reguladores AMS1117. Um dos reguladores fornece uma saída 3.3 V e o outro 5 V. Ambos reguladores conseguem fornecer até 1 A de corrente de saída. O módulo-fonte possui dois canais de entrada de tensão. Um canal possui um conector Jack P4 para tensões entre 9 – 15 V, enquanto o outro possui um conector USB fêmea para fornecer uma tensão de 5 V.

O sistema fixo é alimentado com uma tensão de 12 V, aplicada no conector Jack P2 do módulo-fonte. Os 12 V de tensão podem ser provenientes de uma fonte conectada à rede elétrica, ou de um controlador solar conectado a um painel solar e uma bateria de 12 V. Já o sistema móvel pode ser alimentado por qualquer carregador de bateria portátil com saída em formato USB de 5 V e mínimo 2 A de corrente.

As seções seguintes descrevem cada um dos blocos que compõem os protótipos desenvolvidos.

C.1 TRANSPORTE DE GASES

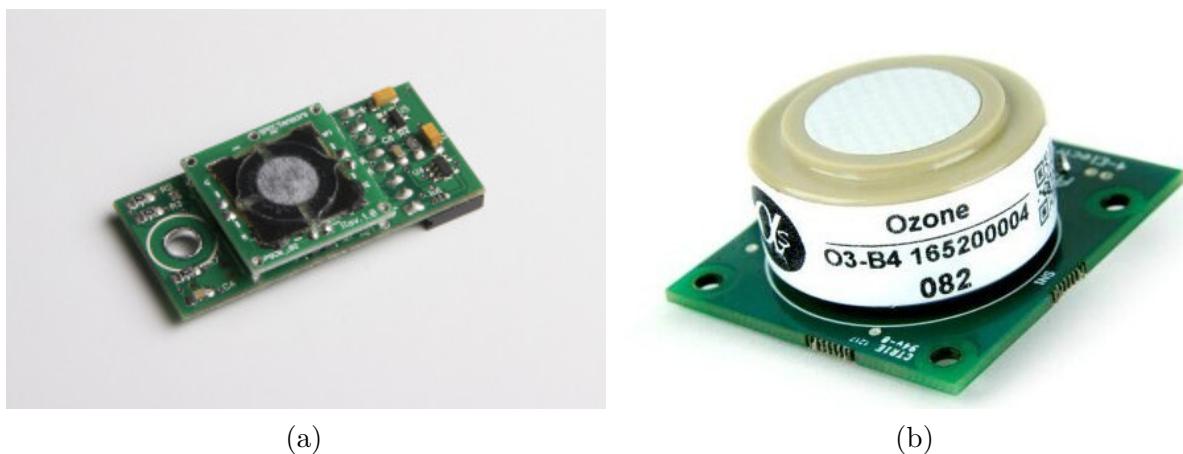
A etapa de transporte de gases é tida como a entrada do sistema. Sua função é capturar amostras do ar no ambiente e direcioná-las para o conjunto de sensores. Nos protótipos desenvolvidos, esta etapa é formada por dois ventiladores de corrente direta e uma câmara de gases. Os ventiladores conduzem o ar desde o ambiente de monitoramento até o interior da câmara. Esta última, por outro lado, consiste em um volume que retém o ar amostrado. No interior dela, os sensores são expostos às porções de ar coletadas para extrair informação de alguns poluentes que podem estar nelas contidos.

As configurações dos ventiladores variam de acordo com a versão do protótipo que os contêm. A versão fixa utiliza ventiladores com tensão nominal de 12 V, e a instalação mecânica deles foi realizada em série para conseguir maior pressão no fluxo do ar. Os ventiladores utilizados na versão móvel, por outro lado, têm uma tensão nominal de 5 V e foram colocados em paralelo. Suas características principais estão dispostas na Tabela 2.

C.2 SENSORIAMENTO

A etapa de sensoriamento consiste em um arranjo de sensores de gases eletroquímicos e os circuitos de condicionamento analógico ou interfaces digitais correspondentes. Os sensores utilizados nos protótipos variam da versão fixa para a móvel, mas de forma geral foram instalados sensores do tipo *screen-printed* fabricados pela SPEC Sensors LLC., e sensores da série B4 da Alphasense Ltd. Os dispositivos de transdução escolhidos são sensíveis aos poluentes regulados na Resolução CONAMA No. 491/2018: CO , NO_2 , SO_2 , O_3 . Além desses gases, foi monitorado também o sulfeto de hidrogênio (H_2S) com um sensor de Alphasense. A modo de ilustração, a Figura 27 mostra os modelos SPEC DGS-CO 968-034 e Alphasense O3-B4, utilizados na medição de CO e O_3 .

Figura 21 – Sensores dos fabricantes a) SPEC e b) Alphasense



A versão fixa dos instrumentos desenvolvidos contém arranjos de sensores das

empresas SPEC e Alphasense. Já o equipamento móvel dispõe apenas de um arranjo de sensores da SPEC Sensors. A seguir são descritas características dos sensores de cada fabricante.

C.2.1 Sensores SPEC

Os sensores da SPEC possuem a configuração característica de três eletrodos (de trabalho, contador e de referência). A sigla SPEC provém do inglês Screen-Printed Electro-Chemical, que é a tecnologia de manufatura utilizada por esse fabricante para produzir seus sensores. Essa tecnologia possibilita fabricar sensores de gases eletroquímicos de alta performance em um encapsulamento fino e de um custo menor que os encapsulamentos mais volumosos, utilizados tradicionalmente para fabricar sensores eletroquímicos (SPEC SENSORS, 2016b). A Figura 21a mostra o sensor SPEC DGS-CO 968-034, utilizado na medição de monóxido de carbono, em sua placa de condicionamento. A Tabela 3 resume as principais características dos sensores que foram utilizados dessa empresa.

C.2.2 Sensores Alphasense

Os sensores da série B4, da Alphasense, utilizam, além dos três eletrodos característicos do princípio de medição eletroquímico, um quarto eletrodo chamado de Eletrodo Auxiliar. Sua função é gerar uma corrente com um valor de intensidade muito próximo ao valor da corrente de fundo do zero (zero background current). Dessa forma é possível compensar a saída dos sensores do efeito desta corrente de zero ou de linha base. A Tabela 4 resume as principais características dos sensores que foram utilizados desse

Tabela 3 – Especificações técnicas dos sensores SPEC

<i>Características</i>	<i>CO</i>	<i>NO₂</i>	<i>SO₂</i>	<i>O₃</i>
Modelo	DGS-CO 968-034	DGS-NO2 968-043	DGS-SO2 968-038	DGS-O3 968-042
Intervalo de medição	0 - 1000 ppm	0 - 5 ppm	0 - 20 ppm	0 - 5 ppm
Resolução	100 ppb	20 ppb	50 ppb	20 ppb
Tensão nominal	3.3 V	3.3 V	3.3 V	3.3 V
Consumo de potência	12 mW	14 mW	12 mW	14 mW
Tempo de resposta*	< 30 s	< 30 s	< 30 s	< 30 s
Temperatura de operação	-20 – 40 °C	-20 – 40 °C	-20 – 40 °C	-20 – 40 °C
Umidade relativa de operação	15 – 95 %	15 – 95 %	15 – 95 %	15 – 95 %

Tabela 4 – Especificações técnicas dos sensores Alphasense

<i>Características</i>	<i>CO</i>	<i>NO₂</i>	<i>SO₂</i>	<i>O₃</i>	<i>H₂S</i>
Modelo	CO-B4	NO ₂ -B43F	SO ₂ -B4	OX-B431	H ₂ S-B4
Intervalo de medição (ppm)	0 - 1000	0 - 20	0 - 100	0 - 20	0 - 100
Resolução (ppb)	4	15	5	15	1
Tempo de resposta (s)*	< 30	< 80	< 60	< 80	< 60
Temperatura de operação (°C)	-30 - 50	-30 - 40	-30 - 50	-30 - 40	-30 - 50
Umidade relativa de operação (%)	15 - 90	15 - 85	15 - 90	15 - 85	15 - 90

fabricante.

C.3 CONDICIONAMENTO

A configuração mais utilizada nos circuitos de condicionamento dos sensores eletroquímicos é o potencióstato. Este circuito controla o potencial aplicado ao eletrodo de trabalho e converte a corrente desse eletrodo em um valor de tensão.

A empresa Alphasense disponibiliza para os sensores da série B4 uma placa de condicionamento chamada de *Individual Sensor Board* (ISB). Esta placa transforma o sinal de corrente de saída do sensor em um sinal de tensão proporcional ao valor de concentração do gás. A SPEC Sensors, por sua vez, disponibiliza uma placa com um microcontrolador dedicado, que condiciona a saída do transdutor e entrega o dado de concentração mediante uma interface digital serial.

C.3.1 Interface de condicionamento dos sensores Alphasense: A Placa de Sensoriamento Individual (ISB)

As placas ISB da Alphasense possuem circuitos de potencióstato compatíveis com a família de sensores B4, de quatro eletrodos. Nesses circuitos, tanto o eletrodo de trabalho quanto o auxiliar possuem etapas de amplificação equivalentes. As tensões de saída destes dois eletrodos são disponibilizados em um conector Molex de 6 vias junto com os canais para a alimentação da placa. A tensão de alimentação das placas ISB pode ser entre 3.5 e 6.4 VDC; nos protótipos desenvolvidos foi utilizada uma tensão de 5 VDC.

O diagrama ilustrado na Figura 22a apresenta as conexões realizadas entre a plataforma Arduino e os sensores da Alphasense através das placas ISB. Percebe-se que cada conjunto composto por um sensor e seu respectivo circuito de condicionamento, ocupa duas entradas analógicas do microcontrolador; uma entrada para o eletrodo auxiliar (AE) e outra para o eletrodo de trabalho (WE). No total foram utilizadas os canais analógicos A0 – A11.

C.3.2 Interface de condicionamento dos sensores SPEC

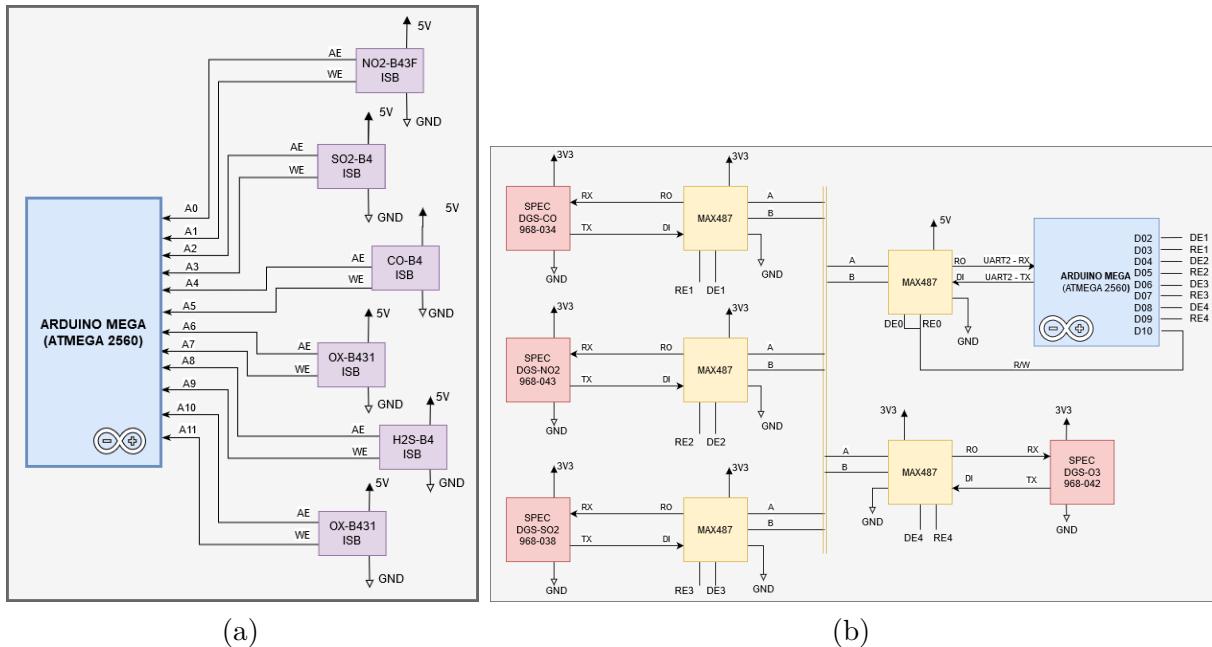
Assim como os sensores da Alphasense, os sensores da empresa SPEC também utilizam um circuito de condicionamento de potenciómetro. Na mesma placa de condicionamento, a SPEC tem incorporado um microcontrolador dedicado e sensores de temperatura e umidade. O microcontrolador converte o valor de tensão de saída do potenciómetro em um valor digital de concentração de gás, e realiza uma compensação, por software, dos efeitos da temperatura e a umidade na medição.

O kit de condicionamento SPEC funciona como uma camada de abstração no que diz respeito ao tratamento e condicionamento das informações, garantindo uma fácil integração com os sistemas de monitoramento. O dispositivo disponibiliza as informações de data e hora, o valor de concentração em ppm/ppb e as leituras de temperatura e umidade através de uma interface serial UART. De igual modo, podem ser realizadas operações como calibração, ajuste de zero e span, configuração dos sensores, e seleção de modo de operação de baixo consumo de energia, através de uma biblioteca com comandos pré definidos (SPEC SENSORS, 2017).

A Figura 22b apresenta um diagrama da conexão do arranjo de sensores da SPEC à plataforma Arduino Mega. Os sensores e o microcontrolador ATMega2560 são acoplados a um barramento RS-485 mediante o transceptor MAX487. Esse transceptor provê uma interface entre os meios de comunicação serial UART e RS-485. O barramento RS-485 consiste basicamente em dois fios A e B que fornecem o meio físico para a transmissão de níveis de tensão que representam os dados seriais enviados pelos diferentes dispositivos. O nível do sinal transmitido através do barramento é determinado pela tensão diferencial entre os conectores A e B, independentemente da voltagem de alimentação dos dispositivos conectados. Como mostra a figura, os transceptores dos sensores são alimentados com uma tensão de 3.3 VDC, enquanto o transceptor do Arduino é alimentado pelo mesmo sinal de 5 VDC que o microcontrolador.

É possível conectar múltiplos dispositivos a um mesmo barramento RS-485 (máximo até 128), sendo necessária a ação de um controlador que determine quem acessa o meio físico a cada instante, para evitar colisões. O microcontrolador ATMega2560 realiza essa função através das saídas digitais D02 – D10. Esses sinais digitais controlam o estado das entradas RE_i e DE_i de cada MAX487, a fim de habilitar/desabilitar cada transceptor para operações de escrita/leitura.

Figura 22 – Interface entre os sensores e o microcontrolador Arduino. a) Alphasense, b) SPEC



Fonte: Desenvolvido pelo autor (2023)

C.4 MICROCONTROLADOR

A etapa de processamento engloba todas as funcionalidades de controle, temporização, geolocalização, aquisição dos dados dos sensores, comunicação e armazenamento dos dados. Todas essas funções são gerenciadas pelo microcontrolador ATMega2560 da Microchip, embarcado na plataforma Arduino Mega 2560. Mais detalhes sobre o firmware desenvolvido para o controle da etapa de processamento são abordados no Apêndice E. A continuação descrevem-se cada um dos módulos que compõem esta etapa.

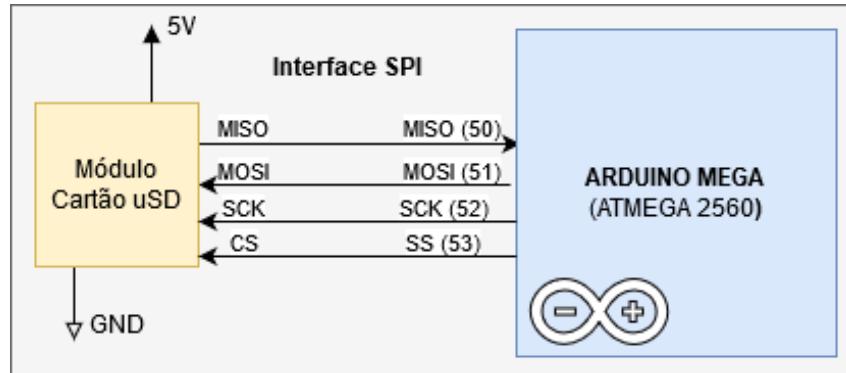
C.4.1 Armazenamento dos dados

Para o armazenamento dos dados foi utilizado um módulo para fazer leitura e escrita diretamente em um cartão micro SD. A comunicação é feita por meio de uma interface SPI, conforme se mostra na Figura 23. O nível de sinal é de 3.3V, mas o módulo possui divisores de tensão nos seus pinos que possibilitam uma ligação direta com placas que trabalham com 5 V, como o Arduino. O módulo é alimentado com uma tensão de 5 V, e suporta cartões Micro SD e Micro SDHC de alta velocidade.

C.4.2 Controle de data e hora e geolocalização

Para manter o controle da data e hora do sistema fixo, e assim acrescentar informação temporal às leituras de gases, foi utilizado o módulo de Relógio de Tempo

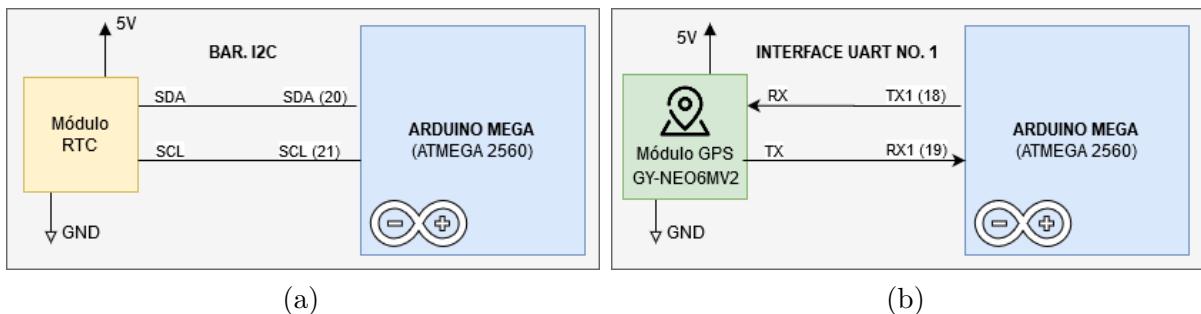
Figura 23 – Interface entre o módulo cartão micro SD e o microcontrolador



Fonte: Desenvolvido pelo autor (2023)

Real (RTC) DS1307. O DS1307 é um relógio/calendário de baixo consumo de potência que utiliza um barramento I^2C bidirecional para a transferência de dados desde (e para) o microcontrolador. O relógio/calendário provê informação de segundos, minutos, horas, dia, mês e ano, incluindo ajuste automático de ano bissexto e de meses com menos de 31 dias. O DS1307 é alimentado por uma tensão de 5 V, e também possui um circuito que detecta falhas de energia e automaticamente aciona a alimentação através de uma bateria. Quando isso sucede, o relógio/calendário mantém a contagem do tempo em um modo de baixo consumo (consumo de corrente menor que 500 nA), estendendo o tempo de vida útil da bateria. A Figura 24a mostra como é realizada sua conexão ao microcontrolador no sistema desenvolvido.

Figura 24 – Interface entre o microcontrolador e os módulos a) RTC e b) GPS



Fonte: Desenvolvido pelo autor (2023)

Na versão móvel, ambos os controles da data e hora e geolocalização são realizados por um mesmo dispositivo GPS, o módulo NEO6MV2. O NEO6M é um receptor GPS de baixo consumo de potência e pequenas dimensões que o tornam uma opção interessante para dispositivos móveis. O módulo possui uma antena integrada, com precisão de aproximadamente 5 metros, e tecnologia para supressão de congestionamentos na comunicação e interferências. A conexão entre o módulo e a plataforma Arduino é realizada através de um barramento serial UART a uma taxa de transferência padrão de 9600 bauds (Figura 24b). Ele pode ser alimentado com uma tensão de 3.3 ou 5 V e seu

consumo de corrente em pleno funcionamento chega a 45 mA. Seus pinos de entrada são compatíveis com níveis de tensão TTL e suportam tensões tanto de 5 como de 3.3 V, independentemente da tensão de alimentação.

C.4.3 Comunicação Wi-Fi

Para a comunicação Wi-Fi é utilizado o módulo ESP-01. Esse módulo incorpora o sistema integrado em um único chip (SoC, System on Chip) ESP8266EX, da empresa Espressif, e uma antena embarcada com ganho de potência de 3dBi, garantindo um alcance de até 90 metros em espaços abertos. O SoC ESP8266EX integra um processador de 32 bits, o Tensilica L106, que implementa os protocolos TCP/IP e o 802.11 b/g/n WLAN MAC. Ele possui como vantagens um baixo consumo de energia atrelado a uma velocidade de clock de 80 MHz. Sua memória RAM, disponível em aplicações em que o sistema está configurado como estação é de aproximadamente 36 kB. O módulo ESP-01 disponibiliza, para armazenar o programa de usuário, uma memória FLASH de 1MB externa que pode ser acessada por um barramento SPI. O módulo disponibiliza quatro portas digitais que são utilizadas principalmente para programar a FLASH de usuário e uma porta serial UART.

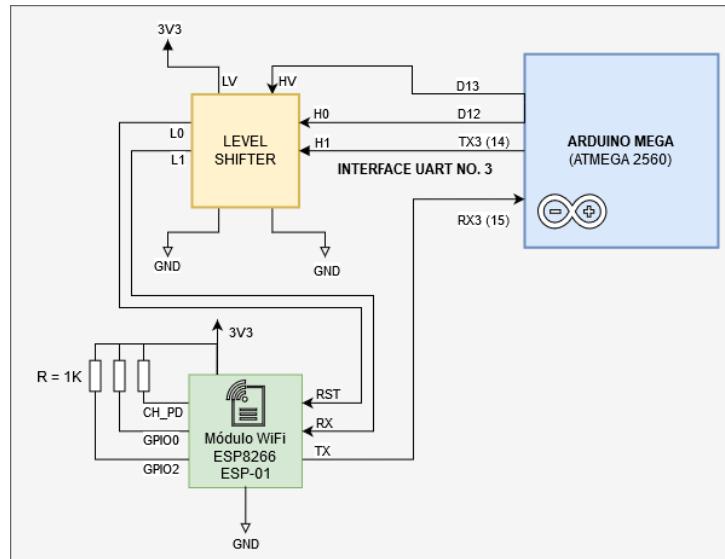


Figura 25 – Interface entre o microcontrolador e o módulo de comunicação Wi-Fi

Fonte: Desenvolvido pelo autor

A Figura 25 apresenta as conexões realizadas entre o ESP-01 e a plataforma Arduino. O módulo opera com uma tensão de 3.3 V, por esse motivo é utilizado um circuito intermediário, um elevador de nível (level shifter) para converter os níveis de tensão de 5 V para 3.3 V, e vice-versa. O pino CH_PD corresponde ao chip enable do ESP-01 e deve ser conectado a um resistor de *pull-up* de 1 kΩ, assim como as entradas GPIO0 e GPIO2. Essas entradas são utilizadas para configurar o ESP8266 em modo gravação (para gravar o programa de usuário) ou modo estação. A figura mostra a configuração do modo

estaçao, com ambas entradas conectadas à 3.3 V por meio de resistores de *pull-up* de 1 kΩ. O pino de entrada RST tem como função reiniciar o módulo. Como esse pino é “ativo baixo”, cada vez que uma tensão de 0 V for aplicada nessa porta o módulo será reiniciado. No circuito desenvolvido, o Arduino pode reiniciar o ESP8266 através da saída digital D12. Já a saída D13 do microcontrolador Arduino é encarregada de manter uma tensão de referência de 5 V no elevador de nível para possibilitar a conversão dos níveis de tensão.

C.5 MONTAGEM DO PROTÓTIPO FIXO

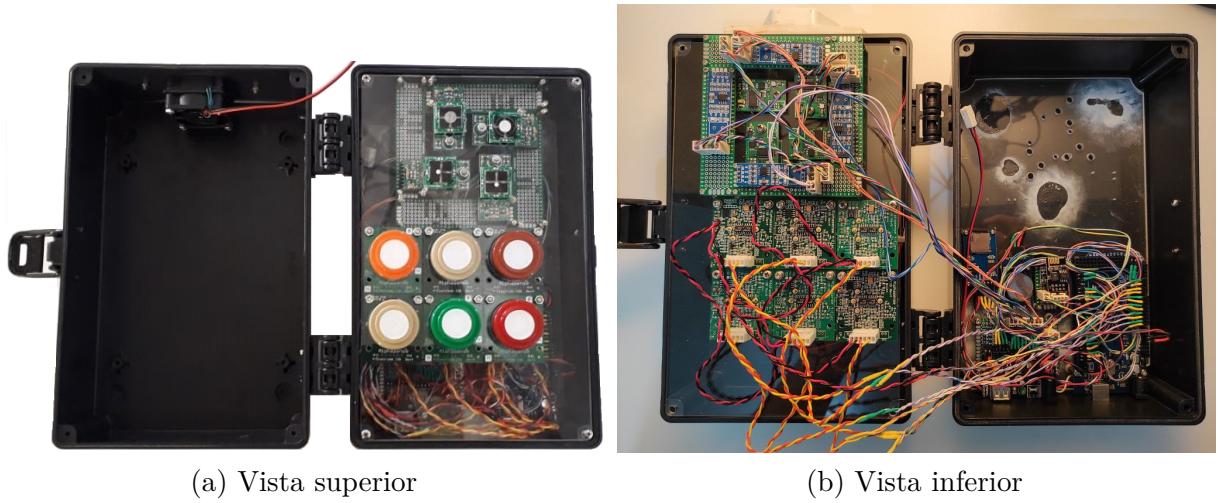
A seguir descreve-se brevemente a montagem e interligação dos elementos de hardware que compõem o protótipo de medição fixa. A Figura 26 mostra o protótipo de monitor fixo instalado em campo. O quadro externo é a caixa ambiental modelo Atlantic 352 00 da Cemar & Legrand com nível de proteção IP66.

Figura 26 – Instalação em campo do protótipo fixo



As Figuras 27a e 27b mostram o módulo de sensoriamento, que é a parte fundamental de todo o sistema. Nele são contidos todos os elementos que compõem o sistema e que foram descritos anteriormente.

Figura 27 – Vista interior do protótipo fixo



Um sistema de transporte de gases, composto por duas ventoinhas de 12VDC, coleta amostras do ar ambiente para dentro da câmara. A entrada é composta por uma flange de 50 mm de diâmetro (que serve para acoplar a câmara no restante do sistema de transporte de gases) e um filtro de tecido. As dimensões das ventoinhas são 40x40mm, e foram fixadas com quatro parafusos M2x30mm com porca e arruela. Dentro do volume da câmara, as superfícies dos sensores de gás interagem com os componentes gasosos e produzem um sinal de resposta proporcional à concentração do gás. A Tabela 5 resume os sensores e placas de condicionamento que foram utilizados nessa versão do equipamento.

Uma placa de acrílico foi utilizada para fixar os sensores de maneira correta e isolá-los do hardware do fluxo de ar. As conexões elétricas para levar os sinais de saída dos sensores até o Arduino foram feitas com fios de seção 0,2mm², soldando “headers” nas pontas e isolando-as corretamente com duto termoretrátil. As Figuras 28a e 28b ilustram respectivamente diagramas de conexão da alimentação elétrica e dos eletrodos dos sensores ao Arduino.

Figura 28 – Diagrama de conexões do conjunto de sensores Alphasense

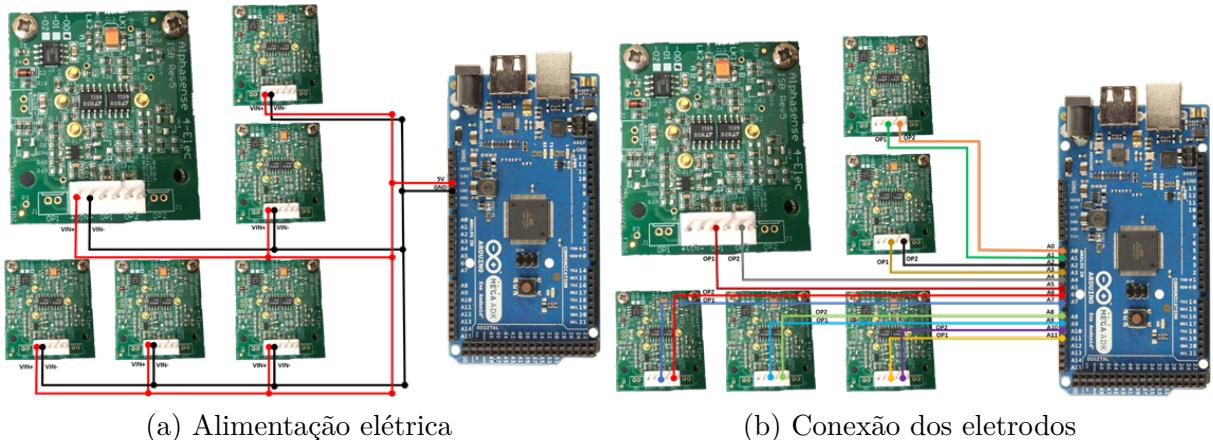


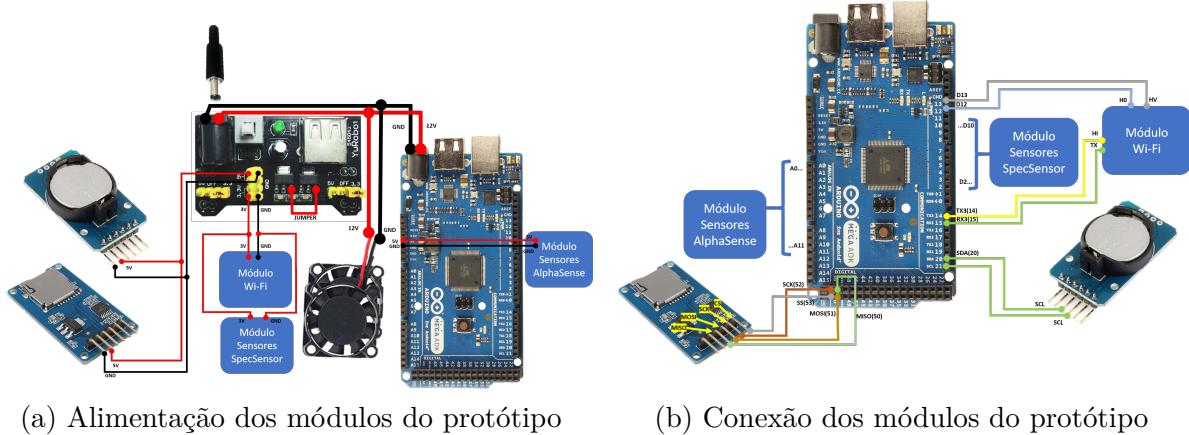
Tabela 5 – Lista de sensores utilizados no protótipo fixo

Qtd.	Ítem	Descrição	Fabricante
1	CO-B4	Sensor de <i>CO</i>	Alphasense
1	H2S-B4	Sensor de <i>H₂S</i>	Alphasense
1	SO2-B4	Sensor de <i>SO₂</i>	Alphasense
1	NO-B4	Sensor de <i>NO</i>	Alphasense
1	NO2-B43F	Sensor de <i>NO₂</i>	Alphasense
1	OX-B431	Sensor de <i>O₃</i>	Alphasense
1	NH3-B1	Sensor de <i>NH₃</i>	Alphasense
3	CO/H2S/SO2 4-electrodes ISB	Placa de condicionamento para sensores da série B4 que medem <i>CO</i> , <i>H₂S</i> e <i>SO₂</i>	Alphasense
1	NO 4-electrodes ISB	Placa de condicionamento para sensores da série B4 que medem <i>NO</i>	Alphasense
1	NO2/O3 4-electrodes ISB	Placa de condicionamento para sensores da série B4 que medem <i>NO₂</i> e <i>O₃</i>	Alphasense
1	NH3 4-electrodes ISB	Placa de condicionamento para sensores da série B4 que medem <i>NH₃</i>	Alphasense
1	DGS-O3-968- 042_9-6-17	Sensor de <i>O₃</i> para IoT	SPEC Sensors
1	DGS-SO2-968- 038	Sensor de <i>SO₂</i> para IoT	SPEC Sensors
1	DGS-NO2-968- 043-9-6-17	Sensor de <i>NO₂</i> para IoT	SPEC Sensors
1	DGS-CO-968- 034	Sensor de <i>CO</i> para IoT	SPEC Sensors

A fixação dos sensores *SPEC* na placa de acrílico mencionada anteriormente foi feita através de placas de prototipagem confeccionadas artesanalmente. Nas placas foram soldados “headers” fêmeas encima dos quais os sensores foram montados. Nas placas também foram instalados os transceptores MAX487 que criam o barramento RS-485 para a conexão serial com o microcontrolador da placa Arduino. As placas são conectadas ao barramento através de fios e conectores do tipo MOLEX. As placas de prototipagem com os sensores foram fixadas diretamente à placa de acrílico com espaçadores M2.

Após a montagem do conjunto de sensores e prefixação dos componentes eletrônicos da câmara de medição, foi realizada a ligação elétrica de alimentação e comunicação de todos os componentes envolvidos no sistema. O diagrama de alimentação é mostrado na Figura 29a. Vale salientar que a alimentação da placa Arduino foi feita diretamente com 12V com fios soldados no conector P2 de entrada. Já a conexão do restante dos módulos com a placa Arduino é ilustrada na Figura 29b.

Figura 29 – Diagrama de conexões do conjunto de sensores Alphasense



C.6 MONTAGEM DO PROTÓTIPO MÓVEL

O equipamento mede poluentes da legislação ambiental brasileira (BRASIL. MINISTÉRIO DO MEIO AMBIENTE (MMA). CONSELHO NACIONAL DO MEIO AMBIENTE (CONAMA), 2018) que são: CO , NO_2 , SO_2 , O_3 e H_2S . Para isso utiliza um conjunto de quatro sensores do fabricante SPEC Sensor que contemplam a medição desse poluentes. O controle da estação de monitoramento, armazenamento e envio de dados é baseado na plataforma Arduino Mega 2560, que utiliza o microcontrolador ATMega2560 da Microchip. Para operar com sucesso, o sistema inclui: módulo Wi-Fi, módulo de cartão SD, módulo GPS e indicadores LED operacionais.

APÊNDICE D – A PLACA CLEAN ARDUINO MEGA

A continuação são descritos os principais módulos que compõem a placa CLEAN Arduino Mega e os protótipos desenvolvidos. A Tabela 6 mostra os principais componentes de hardware utilizados sem considerar os sensores.

D.1 MÓDULO DE SENSORIAMENTO

D.1.1 Sensores

Nos sistemas desenvolvidos foram utilizados sensores de gases eletroquímicos dos fabricantes *SPEC Sensors* e *Alphasense*.

D.1.1.1 Sensores SPEC.

Os sensores da *SPEC* são sensores amperométricos, constituídos por células eletroquímicas (veja Apêndice B) de três eletrodos, i.e.: eletrodos de trabalho, de referência e eletrodo contador. *SPEC* significa *Screen-Printed Electrochemical*, que é a tecnologia utilizada para a fabricação dos sensores, reduzindo seus custos e dimensões, e ainda mantendo um alto desempenho (*SPEC SENSORS*, 2016b).

D.1.1.2 Sensores Alphasense.

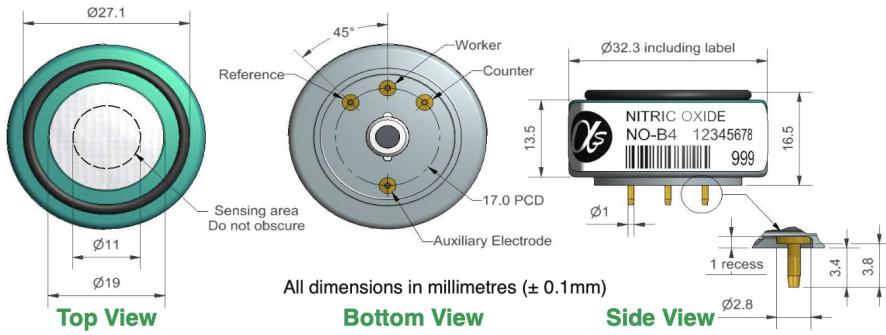
Alphasense fabrica sensores eletroquímicos amperométricos. Especificamente, os sensores da série B4 foram selecionados para os monitores desenvolvidos, já que são indicados pelo fabricante para a medição de baixas concentrações de gases. Estes sensores incorporam um quarto eletrodo, denominado eletrodo auxiliar, que compensa os efeitos da temperatura e da umidade relativa nas leituras dos sensores (BARON; SAFFELL, 2017). A Figura 30 ilustra um sensor Alphasense da série B4, suas dimensões e disposição dos eletrodos. Para mais informações sobre os efeitos das variáveis ambientais nas respostas dos sensores, confira a nota de aplicação AAN 110 da Alphasense. Para informações adicionais sobre os sensores da série B4 da Alphasense, como especificações elétricas, dimensões e pinagem, consulte as fichas técnicas dos modelos de sensores listados na Tabela 1.

Sensores eletroquímicos amperométricos produzem uma corrente de saída que é proporcional à concentração do gás. Para ler este sinal elétrico com um sistema de aquisição de dados, a corrente de saída deve ser transformada em um sinal de tensão. Para isso, o circuito mais utilizado é o potenciómetro. *Alphasense* e *SPEC* fornecem placas de circuito potenciómetro para acoplar facilmente seus sensores a um sistema de monitoramento.

Tabela 6 – Principais componentes utilizados nos dispositivos CLEAN

Item	Descrição	Modelo e fabricante
Arduino Mega 2560	Placa microcontroladora baseada no microcontrolador Microchip ATmega2560	Arduino MEGA 2560 Rev3, de Arduino
DS3231 RTC	Relógio em tempo real (RTC) I2C com oscilador de cristal compensado por temperatura integrado	DS3231, by Maxim Integrated
Soquete micro SD	Soquete TF / micro SD tipo PUSH-PUSH	KLS1-TF-007, da KLS Electronic
Buffer CI 74XX125	Portas de buffer de barramento quádruplas com saídas de 3 estados para buffer de pinos de cartão micro SD	74HC125, da Texas Instruments
Cartão MicroSD	Cartão Micro SD Classe 10 de 16 GB	microSDHC SanDisk Ultra, da SanDisk
Módulo GPS	Módulo GPS NEO-6M c/ antena	GY-GPS6MV2, por u-blox
Módulo GPRS	Shield Arduino - GSM GPRS SIM900 com antena Quad Band	SIM900, da SIMCom
Módulo Wi-Fi	Módulo serial Wi-Fi ESP-01 ESP8266	ESP8266, da Expressif
Sensor BMP280	Sensor digital de temperatura e pressão BMP280 I2C	BMP280, da Bosch Sensortec
Sensor SHT20	Sensor de umidade e temperatura SHT20 I2C	SHT20, por Sensirion
MAX487	Transceptor RS-485 MAX487 Baixa potência, taxa de variação limitada	MAX487, da Maxim Integrated

Figura 30 – Sensor de Monôxido de Nitrogênio Alphasense da série B4



Fonte: (ALPHASENSE, 2019b).

D.1.1.3 Interface de condicionamento de sensores SPEC

Os sensores digitais para IoT fornecidos pela SPEC são compostos por um transdutor eletroquímico montado em uma placa com um circuito potencioscópico, que converte a saída do sensor (corrente elétrica) a tensão. Os sensores incorporam também um microcontrolador e um sensor de temperatura e umidade relativa. O microcontrolador adquire o sinal de tensão do potencioscópico como valores de concentração de gás e realiza uma compensação em software para reduzir os efeitos da temperatura e a umidade relativa nas leituras do sensor. Os valores de concentração, temperatura e umidade relativa são transmitidos através de uma interface UART seguindo um protocolo serial definido pelo fabricante. Esta placa de condicionamento atua como uma camada de abstração para condicionamento de sinal que permite a fácil integração dos sensores a qualquer sistema de monitoramento. Para obter mais informações sobre os sensores SPEC, como especificações elétricas, dimensões, pinagem e protocolo serial, verifique as fichas técnicas dos sensores (Tabela 1) e do kit de desenvolvimento do sensor de gás digital 968-045 (REF).

D.1.1.4 Interface de condicionamento de sensores Alphasense

A Alphasense fornece placas de sensores individuais para seus sensores de gás de 4 eletrodos da série B4 (REF). Essas placas incorporam circuitos potencioscópicos equivalentes para os eletrodos de trabalho e auxiliar. As saídas de cada canal do potencioscópico foram conectadas às entradas analógicas do microcontrolador Arduino MEGA, conforme mostrado na Figura A.3. Os sinais AE e WE representam os sinais correspondentes ao eletrodo auxiliar e de trabalho respectivamente. Seis sensores foram utilizados no protótipo, sendo utilizadas assim doze entradas analógicas do microcontrolador (A0 – A11). Cada módulo ISB foi alimentado com uma tensão de 5 V. Para obter mais detalhes sobre a montagem e conexão dos sensores Alphasense, consulte o Guia de montagem de sensores Alphasense.

D.2 O MICROCONTROLADOR

O microcontrolador Arduino MEGA 2560 coordena as tarefas associadas à aquisição e armazenamento de dados, temporização, geolocalização e comunicação. O firmware para este protótipo está disponível no repositório de firmware. Para obter detalhes sobre a estrutura do firmware e bibliotecas de firmware, consulte a documentação do firmware.

D.2.1 Armazenamento dos dados

Para armazenamento dos dados foi utilizado um módulo micro SD conectado ao microcontrolador através de uma Interface Periférica Serial (SPI). O cartão micro SD funciona com 3.3 V, mas o módulo inclui buffers e um regulador de tensão que permite conexão direta ao Arduino SPI e fonte de alimentação de 5 V, conforme mostra a figura 6.

D.2.2 Relógio de tempo real

Para monitorar a data e a hora de forma contínua foi utilizado o módulo DS1307 Real-Time Clock (RTC). Este módulo é um relógio/calendário de baixo consumo de energia que fornece informações sobre segundos, minutos, horas, dia, data, mês e ano (REF). A data do final do mês é ajustada automaticamente para meses com menos de 31 dias, incluindo correções para anos bissextos. O DS1307 possui um circuito sensor de energia integrado que detecta falhas de energia e alterna automaticamente para a fonte de backup por meio de uma bateria. A operação de cronometragem continua enquanto a peça opera no modo de baixo consumo de energia da fonte de reserva. O módulo se conecta ao Arduino MEGA através da interface I2C e é alimentado com 5V, conforme mostra a figura 7.

D.2.3 Comunicação Wi-Fi

Para a comunicação Wi-Fi foi utilizado o módulo ESP-01 (Figura 8). Este módulo incorpora o microcontrolador ESP8266 junto com uma antena embarcada com ganho de potência de 3dBi e alcance de até 90 m. O ESP8266 é um System on Chip (SoC), fabricado pela Espressif Systems, que integra o microprocessador Tensilica L106 de 32 bits e implementa os protocolos TCP/IP e 802.11 b/g/n WLAN MAC (REF). O ESP-01 também incorpora uma memória flash de 512 kB para programação, que é acessível ao ESP8266 via SPI. Ele também possui oito pinos que são utilizados para alimentação, conexão à porta serial do ESP8266 e conexão aos quatro GPIOs do ESP8266, conforme mostrado na Figura 8. Para mais detalhes sobre a pinagem do ESP-01 e como programar e conectar este módulo para o Arduino MEGA, consulte o Guia de programação do módulo ESP-01. Uma descrição do firmware que desenvolvemos para o microcontrolador ESP8266 pode ser encontrada em The ESP8266 Firmware.

O módulo ESP-01 fornece a conexão a uma rede Wi-Fi para o Arduino MEGA. Conforme mostrado na Figura 9, um circuito de mudança de nível é necessário para fazer a interface com os pinos do Arduino como resultado das diferentes tensões de operação das placas. A comunicação entre os microcontroladores ATMega2560 e ESP8266 é implementada através de uma interface UART (UART3 na placa Arduino), seguindo um protocolo de comunicação que é descrito detalhadamente no Guia de Programação do Módulo ESP-01. O Arduino atua como mestre do ESP8266, cuja única iniciativa é estabelecer conexão com a Internet. Uma vez estabelecida a conexão, o Arduino pode enviar comandos para criar posts HTTP, obter o horário da internet ou obter as coordenadas de geolocalização do Google; para obter mais detalhes, consulte o Guia de programação do módulo ESP-01. O microcontrolador Arduino também pode redefinir o ESP8266 através do pino D12 GPIO.

APÊNDICE E – O *FIRMWARE CLEAN*

O *firmware* dos dispositivos foi desenvolvido para o microcontrolador Microchip ATMega2560 embarcado em uma plataforma Arduino Mega. O código foi implementado na linguagem de programação C/C++ utilizando o *framework* de Arduino, disponível na IDE PlatformIO para o editor de código Microsoft Visual Studio (VSCode).

Para a programação de todas as funcionalidades do *firmware*, o código foi estruturado em um conjunto de classes. Essa estrutura foi concebida visando seu reaproveitamento em outros microcontroladores suportados no Framework Arduino, como o ESP8266 da Espressif, e também para facilitar a revisão e manutenção do código. As classes desenvolvidas para o projeto estão distribuídas em três pacotes de bibliotecas, mostrados na Figura ???: o pacote *IoT*, o pacote *Data* e o pacote *Hardware Interfaces*.

O pacote *IoT* encapsula os processos associados à conexão na rede Wi-Fi, comunicação com o servidor e envio de dados pelo protocolo *HTTP*. Já o pacote *Data* engloba todas as funcionalidades relacionadas à preparação dos dados dos sensores para seu armazenamento e transmissão. Este pacote permite abstrair as informações de concentração adquiridos pelos sensores de gases, de detalhes específicos sobre o funcionamento e operação do hardware destes sensores. Ele atua como uma camada intermediária entre as tarefas de aquisição, e as de armazenamento e transmissão dos dados. Por último, o pacote *Hardware Interfaces* agrupa as classes e estruturas utilizadas para interfacear todo o hardware periférico ao microcontrolador utilizado, como sensores, módulos de temporização, módulos de geolocalização e módulos de armazenamento.

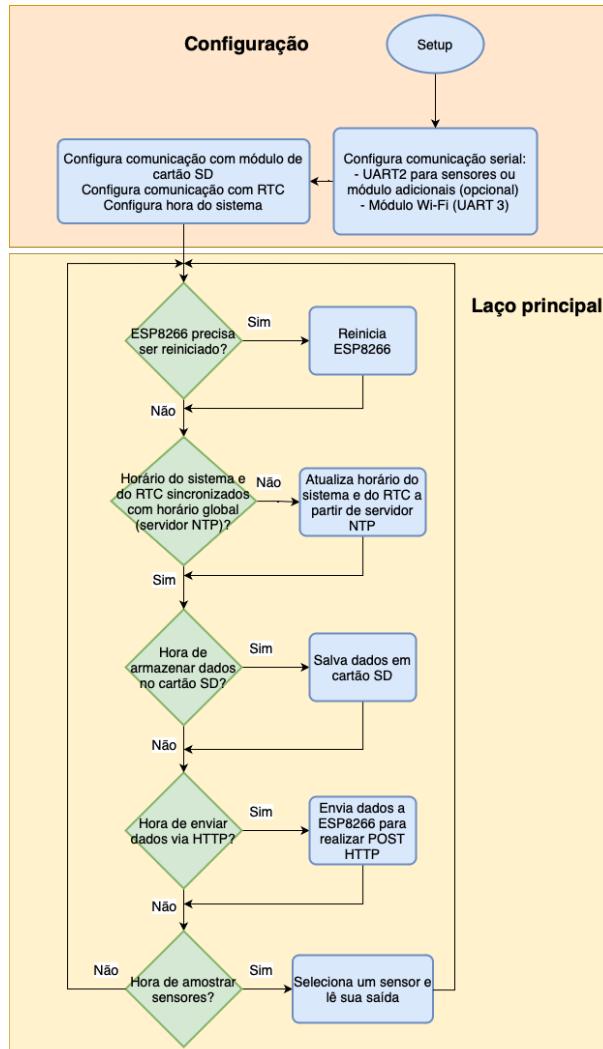
E.1 CÓDIGO CLEAN ARDUINO MEGA

O código consiste em duas partes: uma para configuração (*setup*) e outra para o laço (*loop*) de execução principal. A versão atual do código possui quatro funcionalidades principais que estão de acordo com o hardware do monitor, sendo elas:

1. Amostragem, ou leitura das saídas dos sensores
2. Armazenamento das informações de concentração de gás em um cartão SD
3. Envio dos dados dos sensores para o servidor Renovar através do microcontrolador ESP8266
4. Leitura das coordenadas geográficas do local onde ocorre cada leitura

A Figura 31 mostra um fluxograma do código programado para o ATMega2560. Como acontece com todo programa do Arduino Framework, o código é executado em duas funções principais: `setup()` e `loop()`. Na versão atual do firmware, o `setup` prepara a comunicação entre os módulos externos (i.e.: RTC, Wi-Fi, cartão de memória e sensores

Figura 31 – Fluxograma do firmware programado para o microcontrolador Arduino MEGA



Fonte: Desenvolvido pelo autor (2023)

seriais) e o microcontrolador. A função `loop()` verifica se o ESP8266 não responde há algum tempo, e, se for o caso, é enviado um sinal RESET para o ESP8266. O resto da função está dividido em quatro seções que são executadas periodicamente e controlam as funcionalidades mencionadas.

O firmware inclui mais uma funcionalidade que está separada do fluxo principal do programa: a função `serialEvent3()` que trata os eventos de interrupção da porta serial UART3. Ela é executada cada vez que novos dados são recebidos no buffer dessa interface de hardware que estabelece a comunicação entre o Arduino Mega e o microcontrolador ESP8266. A seguir, são descritas as diferentes seções do código.

E.1.1 Identificação do dispositivo e seus sensores

Uma parte crucial do firmware é a identificação do dispositivo e dos sensores conectados a ele. Essas identificações deverão corresponder às que tenham sido previamente configuradas no servidor da aplicação Web Renovar, já que serão utilizadas pela aplicação backend para atualizar o banco de dados. A definição do dispositivo é feita na seguinte linha:

```
unsigned long Device::id = <THE NUMBER OF YOUR DEVICE>;
```

Depois disso, devem ser definidos os IDs dos sensores, de forma que cada um deles represente uma variável na aplicação Renovar. O código da Lista E.1.1 exemplifica a definição dos identificadores de 6 sensores previamente registrados na aplicação Renovar.

Listing E.1 – Definição dos identificadores dos sensores de um dispositivo

```
enum iotId_e {
    // The ID of the CO gas concentration read from a sensor
    CO_ID      = 156,
    // The ID of the NO2 gas concentration read from a sensor
    NO2_ID     = 157,
    // The ID of the O3 gas concentration read from a sensor
    O3_ID      = 158,
    // The ID of the SO2 gas concentration read from a sensor
    SO2_ID     = 159,
    // The ID of the temperature variable
    TEMP_ID    = 160,
    // The ID of the relative humidity variable
    RHUM_ID    = 161
};
```

E.1.2 Configuração: a função setup()

A função `setup()` prepara a comunicação entre os módulos externos e o microcontrolador. O código desta função é mostrado na Lista E.1.2. Primeiramente o programa configura as portas seriais que serão utilizadas para comunicação com os sensores seriais utilizando uma interface RS-458 (RS485_2) e ESP8266 (Serial3). Cada porta serial é inicializada a uma taxa de transmissão previamente definida no código, conforme será descrito posteriormente. A porta serial UART0 (Serial) é usada para depuração. O RS485_2 implementa uma interface serial ao barramento RS485 que conecta os sensores ao microcontrolador. A função também reinicia o microcontrolador ESP8266 através do objeto `espIoT`, inicializa as interfaces para o cartão SD e módulos RTC e define a hora do sistema. A continuação são descritos os objetos, constantes e funções usados nesta parte do código.

Listing E.2 – Definição dos identificadores dos sensores de um dispositivo

```

void setup ()
{
    Serial . begin (9600);
#ifndef FIXED_DEVICE
    Serial1 . begin (GPSBaud);
#endif
    Serial3 . begin (9600UL);

    pinMode (13 , OUTPUT);
    digitalWrite (13 , HIGH);

    espIoT . restart ();

    SD . begin (CHIPSEL_PIN);

    Rtc . Begin ();
    if (!Rtc . GetIsRunning ())
    {
        Rtc . SetIsRunning (true);
    }
#ifndef DS3231
    Rtc . Enable32kHzPin (false);
    Rtc . SetSquareWavePin (DS3231SquareWavePin_ModeNone);
#endif
    time_t now = Rtc . GetDateTime () . Epoch32Time ();
    setSyncProvider (sync_time);
    setSyncInterval (5*SECS_PER_MIN);
    TimeDriver :: config (TIMEZONE_SEC);
    SHT20 . initSHT20 ();
    BMP280 . begin (BMP280_ADDRESS_ALT, BMP280_CHIPID);
    Alpha_OPc . begin ();
#ifndef FIXED_DEVICE
    GPSDriver :: set_coordinates (true,
                                DeviceFixedLocation :: LATITUDE,
                                DeviceFixedLocation :: LONGITUDE,
                                DeviceFixedLocation :: ALTITUDE);
#endif
    mLastTime = millis ();
}

```

```

mLastTimeGPS = mLastTime;
mLastTimeSD = mLastTime;
mLastTimeHTTP = mLastTime;
}

```

E.1.2.1 Serial, Serial1, Serial3

Esses objetos, declarados no *framework* Arduino, representam as portas UART do microcontrolador. Os objetos são inicializados pela função `begin()`, que recebe a taxa de transmissão da comunicação serial. O objeto `Serial` representa a porta serial `UART0` que é usada para depuração do programa. Já os objetos `Serial1` e `Serial3`, que representam as portas seriais `UART1` e `UART3` do Arduino Mega, são utilizadas para comunicação com o módulo GPS e o microcontrolador `ESP8266` respectivamente.

A variável `GPSBaud` é uma constante que define a velocidade em *bits* por segundo da comunicação serial entre o microcontrolador `ATMega` do Arduino e o módulo GPS. O valor predeterminado é 9600 baúdios, definido na biblioteca `serial-geo-interface`, mas pode ser redefinido segundo a aplicação.

E.1.2.2 espIoT

Este é um objeto da classe `ESPSerialInterface` definido na biblioteca `serial-internet-interface`. Este objeto controla a comunicação com o `ESP8266` conectado ao `UART3` do Arduino. A função `restart()` envia um sinal de *RESET* para o `ESP8266`. O objeto `espIoT` é definido no código da seguinte forma: `ESPSerialInterface espIoT(&Serial3);`

E.1.2.3 SD

Este é um objeto do `SDClass` declarado no núcleo do Arduino para interfacear módulos de cartão SD. Ele é inicializado com um método `begin()` que recebe o pino digital que se conecta ao pino CS do módulo. O pino digital utilizado para a versão atual do hardware e firmware é definido no arquivo `hardstorage.h` da seguinte forma: `#define CHIPSEL_PIN 53`

E.1.2.4 Rtc

Este é um objeto da classe `RtcDS3231` definida na biblioteca `Rtc` de Makuna. O objeto `Rtc` é declarado da seguinte forma:

```

#define I2C Wire
RtcDS3231<TwoWire> Rtc(I2C);

```

A instância `Rtc` é inicializada com a função `begin()` e posteriormente o código verifica se o módulo está funcionando por meio de uma chamada ao método `GetIsRunning()`. Caso não esteja rodando, o código chama o método `SetIsRunning(true)`. Caso o módulo RTC DS3231 tenha sido configurado incorretamente, o código também redefine seu status através do seguinte código:

```
#ifdef DS3231
    Rtc.Enable32kHzPin(false);
    Rtc.SetSquareWavePin(DS3231SquareWavePin_ModeNone);
#endif
```

A chamada ao método `Rtc.GetDateTime().Epoch32Time()` da classe `RtcDS3231` obtém a hora atual do módulo RTC no formato UNIX. Já os métodos `setSyncProvider(getExternalTime getTimeFunction)` e `setSyncInterval(time_t interval)` são funções da biblioteca `Time` que permitem a sincronização automática da hora do sistema com uma fonte de relógio determinada. Neste caso, a fonte utilizada para sincronização é o módulo RTC. A função `setSyncProvider()` recebe um ponteiro para uma função que retorna a data e hora atual como uma variável de tipo `time_t`. Neste caso, a função que é passada como ponteiro é `sync_time()`, declarada anteriormente no código conforme mostrado abaixo:

```
RtcDS3231<TwoWire> Rtc(I2C);
RTCDS3231Interface My_RTCInterface(&Rtc);
time_t sync_time() {
    return RTCDriver<RtcDS3231<TwoWire>>::
        sync_time_from_RTC(&My_RTCInterface);
}
```

A função `setSyncInterval()` recebe o período de sincronização da hora do sistema, que neste caso foi definido como 5 segundos.

E.1.3 Interrupção Serial3

O código da função que trata a interrupção do UART3 é mostrado na Lista E.1.3. Cada vez que os dados estiverem disponíveis no *buffer* de entrada da porta serial, o objeto `espIoT` irá analisar a cadeia de caracteres recebida.

Listing E.3 – Código para tratamento da interrupção da porta serial UART3

```
void serialEvent3() {
    if (Serial3.available()) {
        String buffer = Serial3.readStringUntil(';' );
        Serial3.flush();
        espIoT.parse_esp_string(buffer);
```

```

    }
}
```

E.1.4 Laço principal do programa: a função loop()

A sequência de instruções do programa da placa CLEAN Arduino Mega é executado dentro de um laço infinito definido na função `loop()` do *Framework* Arduino. Esta função é responsável por tratar quatro funcionalidades que foram mencionadas anteriormente, i.e.: amostragem, armazenamento, envio de dados e geolocalização. A Lista E.1.4 mostra o código da função `loop()`.

Listing E.4 – Código do laço de execução do programa

```

void loop()
{
    static bool sd_ok = false;

    espIoT.watch_dog();

    if (( ! TimeDriver :: _already_up_to_date ()) ) espIoT.request_time ();
    if (( ! My_RTCInterface .is_up_to_date () ))
        RTCDriver<RtcDS3231<TwoWire>>::update_rtc(&My_RTCInterface , now ());

    // /*
    if (( millis () - mLastTime) >= SAMPLE_ITERATION_PERIOD_MS)
    {
        mLastTime = millis ();
        static uint8_t index = 0;
        Vars [index]→smooth( sensors [index]→read ());
        index = (index >= numSensors - 1) ? 0 : index + 1;
    }
    // */

    // /*
    if (( millis () - mLasteuSD) >= uSD_TIME_MSEC)
    {
        mLasteuSD = millis ();
        static uint8_t data_index_uSD = 0;
        Vars [data_index_uSD]→sense(&data );
```

```

char* filename = (char*)malloc(strlen_P(
    filenames [data_index_uSD])+1);
strcpy_P(filename , filenames [data_index_uSD]);
if(open_file(filename))
    sd_ok = save_to_file(&data , filename );
else SD.begin(CHIPSEL_PIN);
free( filename );

data_index_uSD = (data_index_uSD >= numSensors-1) ? 0 :
    data_index_uSD + 1;
}

// */

if(( millis () - mLastTimeHTTP) >= HTTP_TIME_MSEC)
{
    mLastTimeHTTP = millis ();
    static uint8_t data_index_iot = 0;

    Vars [data_index_iot]->sense(&data );
    readings [0] = &data ;
    if (!espIoT.send_http_post(&data)) print_debug("Couldn't post!");
    data_index_iot = (data_index_iot >= numSensors-1) ? 0 :
        data_index_iot + 1;
}

// */

if(( millis () - mLastTimeGPS) >= MSECS_GPSOUTDATE)
{
    static uint8_t gps_tries = 0;
    print_debug(" [MAIN] GPS");
    mLastTimeGPS = millis ();
    if (!gps.read_gps(MSECS_GPSOUTDATE/2))
    {
        if (gps_tries++ > 7)
        {
            GPSDriver::set_coordinates(true,
                DeviceDefaultLocation::LATITUDE,

```

```

        DeviceDefaultLocation ::LONGITUDE,
        DeviceDefaultLocation ::ALTITUDE);
    gps_tries = 8;
}
}
else {
    gps_tries = 0;
}
//
}
}

```

As quatro funcionalidades principais que o código executa periodicamente são controladas pelas variáveis `mLastTime`, `mLastTimeGPS`, `mLastTimeuSD` e `mLastTimeHTTP`, que armazenam a marca o instante de tempo em que cada funcionalidade é executada. As constantes `uSD_TIME_MSEC`, `HTTP_TIME_SEC`, `SAMPLE_ITERATION_PERIOD_MS` e `MSECS_GPSOUTDATE` representam os períodos em que cada funcionalidade deve ser executada, conforme está resumido na Tabela 7. Em cada ciclo do laço, o objeto `espIoT` chama ao método `watch_dog()` para verificar se existe alguma solicitação enviada ao ESP8266 cujo tempo de espera tenha expirado. Caso isso aconteça, o ESP8266 será reiniciado. A função `loop()` também verifica se o microcontrolador atualizou seu horário a partir de um servidor de data e hora da Internet, caso contrário, uma solicitação é enviada ao ESP8266 para retornar o horário atual da Internet.

Tabela 7 – Constantes e variáveis utilizadas para controlar a execução de cada funcionalidade no firmware

Funcionalidade	Período	Constante definida no código	Variável de controle
Amostragem	6 segundos	<code>SAMPLE_ITERATION_PERIOD_MS</code>	<code>mLastTime</code>
Armazenamento	60 segundos	<code>uSD_TIME_MSEC</code>	<code>mLastTimeuSD</code>
Envio de dados	60 segundos	<code>HTTP_TIME_MSEC</code>	<code>mLastTimeHTTP</code>
Geolocalização	70 segundos	<code>MSECS_GPSOUTDATE</code>	<code>mLastTimeGPS</code>

Fonte: Desenvolvido pelo autor (2023)

O código também verifica se o módulo RTC foi atualizado com o horário da internet. Caso contrário, ele chama o método `update_rtc()` da classe `RTCDriver`. Esta classe é um *template* para controlar as funcionalidades relacionadas a um módulo RTC genérico, como atualizar seu horário, por exemplo. O método `update_rtc()` recebe um ponteiro para uma instância da classe `RTCInterface`, que cria uma interface para o hardware de

qualquer módulo RTC. `RTCInterface` é uma classe abstrata, portanto, para instanciar essa interface em um módulo RTC, uma classe concreta deve ser herdada dele. No presente caso, esta instância foi implementada no objeto `My_RTCInterface`, declarado previamente no código conforme mostrado abaixo.

```
class RTCDS3231Interface : public RTCInterface<RtcDS3231<TwoWire>>
{
public:
    RTCDS3231Interface( RtcDS3231<TwoWire>* rtc ) :
        RTCInterface<RtcDS3231<TwoWire>>( rtc ) {}

    virtual void set_time( time_t t ) {
        RtcDateTime dt;
        dt.InitWithEpoch32Time( t );
        _rtc->SetDateTime( dt );
    }

    virtual time_t get_time() {
        return _rtc->GetDateTime().Epoch32Time();
    }
};

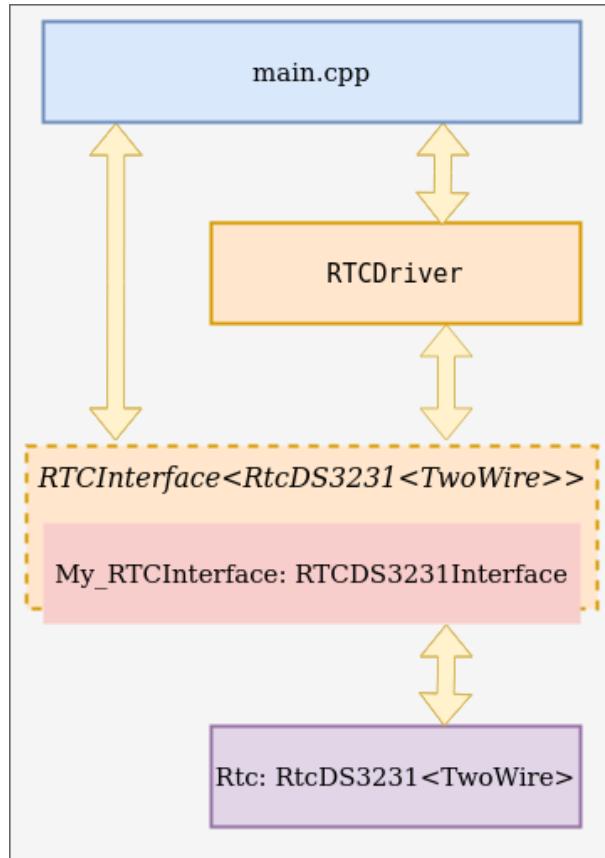
#define I2C Wire
RtcDS3231<TwoWire> Rtc( I2C );
RTCDS3231Interface My_RTCInterface(&Rtc);
```

Como pode ser observado, a classe `RTCDS3231Interface` herda de `RTCInterface`. Quando o objeto daquela classe é declarado, ele recebe em seu construtor uma referência a um objeto `rtc`, que neste caso representa o próprio módulo DS3231. Resumindo, o objeto `rtc` representa o módulo DS3231; o objeto `My_RTCInterface` implementa a interface entre o módulo RTC e o código principal; e a classe `RTCDriver` controla as funcionalidades do módulo dentro do código. Essa relação é representada no diagrama da Figura 32.

E.1.4.1 Leitura dos sensores

A seção do código que lê os sensores de gás é mostrada na Lista E.1.4.1. O código primeiro verifica se o tempo de leitura dos sensores já passou e atualiza a variável `mLastTime`. Esta seção do código basicamente itera entre as listas `Vars` e `sensors` para obter a leitura de cada sensor e atribuir a variável correspondente. A lista `sensors` contém todos os sensores conectados na placa CLEAN, já a lista `Vars` contém as variáveis que cada sensor representa; um representa a camada de *hardware* enquanto o outro representa

Figura 32 – Módulos e interfaces usados para controle e interface do RTC



Fonte: Desenvolvido pelo autor (2023)

uma camada de mais alto nível dos dados, as variáveis físicas. Dessa forma a informação de mais baixo nível contida no sinal de saída dos sensores é transportada para camadas de dados superiores que possibilita seu armazenamento e transmissão remota. Na Lista E.1.4.1 ilustra-se o código que implementa cada uma das listas.

Listing E.5 – Código para leitura dos sensores

```

if (( millis () - mLastTime) >= SAMPLE_ITERATION_PERIOD_MS)
{
    mLastTime = millis ();
    static uint8_t index = 0;
    Vars [index]→smooth( sensors [index]→read ());
    index = (index >= numSensors - 1) ? 0 : index + 1;
}

```

Listing E.6 – Declaração das listas de sensores e variáveis

```

Sensor* sensors [numSensors] =
{
    ( SensorInterface<Adafruit_BMP280>*&)IntTempSensor ,

```

```

( SensorInterface<Adafruit_BMP280>*)&IntPresSensor ,
( SensorInterface<AlphaSenseCompensator>)
    (new AlphaSenseCompensatorSensor(&Alpha_COComp)) ,
( SensorInterface<AlphaSenseCompensator>)
    (new AlphaSenseCompensatorSensor(&Alpha_NO2Comp)) ,
( SensorInterface<AlphaSenseCompensator>)
    (new AlphaSenseCompensatorSensor(&Alpha_SO2_1_Comp)) ,
( SensorInterface<AlphaSenseCompensator>)
    (new AlphaSenseCompensatorSensor(&Alpha_O3_1_Comp)) ,
( SensorInterface<AlphaSenseCompensator>)
    (new AlphaSenseCompensatorSensor(&Alpha_O3_2_Comp)) ,
( SensorInterface<AlphaSenseCompensator>)
    (new AlphaSenseCompensatorSensor(&Alpha_SO2_2_Comp)) ,
( SensorInterface<DFRobot_SHT20>*)&ExtTempSensor ,
( SensorInterface<DFRobot_SHT20>*)&ExtHumSensor ,
( SensorInterface<AlphasenseOPC>)*(new
    AlphaSenseOPCPM10Sensor(&Alpha_OPc)) ,
( SensorInterface<AlphasenseOPC>)*(new
    AlphaSenseOPCPM2_5Sensor(&Alpha_OPc)) ,
( SensorInterface<AlphasenseOPC>)*(new
    AlphaSenseOPCPM1Sensor(&Alpha_OPc)) ,
( SensorInterface<AlphasenseOPC>)*(new
    AlphaSenseOPCTempSensor(&Alpha_OPc)) ,
( SensorInterface<AlphasenseOPC>)*(new
    AlphaSenseOPCHumSensor(&Alpha_OPc)) )
};

Variable* Vars [ numSensors ] =
{
    new Temperature(TEMPERATURE_ID, SI_TEMP_Celsius, BUFFER_SIZE) ,
    new Pressure(PRESSURE_ID, SI_PRES_Pascal, BUFFER_SIZE) ,
    new GasConcentration(ALPHA_CO_ID, SI_CONC_ppb, BUFFER_SIZE) ,
    new GasConcentration(ALPHA_NO2_ID, SI_CONC_ppb, BUFFER_SIZE) ,
    new GasConcentration(ALPHA_SO2_1_ID, SI_CONC_ppb, BUFFER_SIZE) ,
    new GasConcentration(ALPHA_OX_1_ID, SI_CONC_ppb, BUFFER_SIZE) ,
    new GasConcentration(ALPHA_OX_2_ID, SI_CONC_ppb, BUFFER_SIZE) ,
    new GasConcentration(ALPHA_SO2_2_ID, SI_CONC_ppb, BUFFER_SIZE) ,
    new Temperature(EXT_TEMPERATURE_ID, SI_TEMP_Celsius, BUFFER_SIZE) ,
    new Humidity(EXT_HUMIDITY_ID, SI_HUMD_Relative, BUFFER_SIZE) ,

```

```

new GasConcentration (PM10_ID, SI_CONC_ug, BUFFER_SIZE) ,
new GasConcentration (PM25_ID, SI_CONC_ug, BUFFER_SIZE) ,
new GasConcentration (PM01_ID, SI_CONC_ug, BUFFER_SIZE) ,
new Temperature (OPC_TEMPERATURE_ID, SI_TEMP_Celsius, BUFFER_SIZE) ,
new Humidity (OPC_HUMIDITY_ID, SI_HUMD_Relative, BUFFER_SIZE)
};


```

E.1.4.2 Armazenamento dos dados

A seção do código que armazena os dados em um cartão SD é mostrada na Lista . O código primeiro verifica se o tempo de armazenamento dos dados já passou e atualiza a variável `mLastTimeuSD`. Esta seção do código transfere os dados de cada variável para um objeto `data`, que é do tipo `SensorData` e que é usado para armazenar as informações da variável. O método utilizado para transferir as informações de `Variable` para `SensorData` é a função `sense()`, que recebe um ponteiro para `SensorData`. Este objeto `data` é definido anteriormente no código conforme mostrado abaixo. Depois que os dados forem transferidos para a instância de `SensorData`, eles são armazenados em um arquivo no cartão SD.

Listing E.7 – Sequência de armazenamento de dados

```

SensorData data;

// ...

void loop()
{
    // ...
    if(( millis () - mLastTimeuSD ) >= uSD_TIME_MSEC)
    {
        mLastTimeuSD = millis ();
        static uint8_t data_index_uSD = 0;
        Vars [data_index_uSD]→sense (&data );

        char* filename = (char*) malloc (strlen_P (
            filenAMES [data_index_uSD])+1);
        strcpy_P (filename , filenAMES [data_index_uSD] );
        if(open_file (filename ))
            sd_ok = save_to_file (&data , filename );
        else SD.begin (CHIPSEL_PIN);
        free (filename );
    }
}

```

```

    data_index_uSD = (data_index_uSD >= numSensors-1) ? 0 :
                    data_index_uSD + 1;
}
// ...
}

```

E.1.4.3 Envio de dados via protocolo *HTTP*

A seção do código que envia os dados ao ESP8266 para postagem em um servidor HTTP é mostrada abaixo. Assim como as demais seções do código, esta seção primeiro verifica se o tempo de envio dos dados já passou e atualiza a variável `mLastTimeHTTP`. Depois disso, ele utiliza o último objeto `SensorData` armazenado para enviar as informações adquiridas para a variável correspondente. O objeto `espIoT` enviará uma cadeia de caracteres com um objeto JSON contendo as informações a serem postadas pelo ESP8266. O método `send_http_post()` recebe um ponteiro para um `DataContainer`. Como a classe `SensorData` herda de `DataContainer`, cada item nos dados pode ser convertido em um ponteiro desse tipo

```

if (( millis () - mLastTimeHTTP ) >= HTTP_TIME_MSEC)
{
    mLastTimeHTTP = millis ();
    static uint8_t data_index_iot = 0;

    Vars [ data_index_iot ]->sense (&data );
    readings [ 0 ] = &data ;
    if (!espIoT . send _http _post (&data )) print _debug ( " Couldn ' t post ! " );
    data_index_iot = ( data_index_iot >= numSensors -1) ? 0 :
                    data_index_iot + 1;
}

```

E.1.4.4 Geolocalização

Por fim, a continuação mostra a seção de código que atualiza as informações de geolocalização do módulo GPS. Assim como nas outras seções do código, esta seção primeiro verifica se passou o tempo para atualizar as informações do GPS e atualiza a variável `mLastTimeGPS`. Para ler o módulo GPS, o objeto `gps` invoca o método `readGPS()`. Este método toma como parâmetro o tempo máximo que o Arduino deve aguardar uma resposta do módulo GPS, neste caso `MSECS_GPSOUTDATE/2`. O objeto `gps` é uma instância da classe `TinyGPSSerialInterface` que está previamente definida no código conforme mostrado. O construtor deste objeto recebe uma referência à porta serial utilizada para comunicação com o módulo, neste caso `Serial1`. Em caso de falha na comunicação

com o módulo GPS, depois da sétima tentativa, é setado um valor padrão previamente configurado.

```
TinyGPSSerialInterface gps(&Serial1);  
  
// ...  
  
void loop()  
{  
    // ...  
  
    if ((millis() - mLastTimeGPS) >= MSECS_GPSOUTDATE)  
    {  
        static uint8_t gps_tries = 0;  
        print_debug("[MAIN] GPS");  
        mLastTimeGPS = millis();  
        if (!gps.read_gps(MSECS_GPSOUTDATE/2))  
        {  
            if (gps_tries++ > 7) {  
                GPSDriver::set_coordinates(  
                    true,  
                    DeviceDefaultLocation::LATITUDE,  
                    DeviceDefaultLocation::LONGITUDE,  
                    DeviceDefaultLocation::ALTITUDE);  
                gps_tries = 8;  
            }  
        }  
        else {  
            gps_tries = 0;  
        }  
    }  
}
```

APÊNDICE F – O FIRMWARE DO MICROCONTROLADOR ESP8266

O *firmware* do microcontrolador ESP8266 foi desenvolvido na linguagem de programação C/C++ utilizando o *Framework* Arduino. O código foi programado na IDE PlatformIO para o editor de código Microsoft Visual Studio (VSCode). A versão atual do firmware do ESP8266 possui três funcionalidades principais, que são:

1. Prover conexão à Internet a um microcontrolador principal, que atua como "mestre", via uma rede *Wi-Fi*
2. Envio dos dados coletados pelo microcontrolador "mestre" para o servidor web Renovar mediante *POST HTTP*
3. Obter a data e hora atuais de um servidor NTP

A Figura 33 mostra um fluxograma do código programado para o microcontrolador ESP8266. Em primeiro lugar, o programa estabelece uma conexão à Internet através de uma rede *Wi-Fi*. Uma vez estabelecida a conexão, o programa se mantém escutando a conexão serial com o microcontrolador "mestre". Para cada solicitação recebida do "mestre", o ESP8266 executa a operação associada à solicitação e envia seu resultado de volta para o "mestre". As mensagens trocadas entre o ESP8266 e o outro microcontrolador são cadeias de caracteres no formato JSON.

Na versão atual do *firmware*, o "mestre" pode realizar dois tipos de solicitações. Pode solicitar o envio de uma requisição para a API Renovar na forma de POST HTTP com leituras dos sensores ou pode solicitar a data e hora de um servidor NTP.

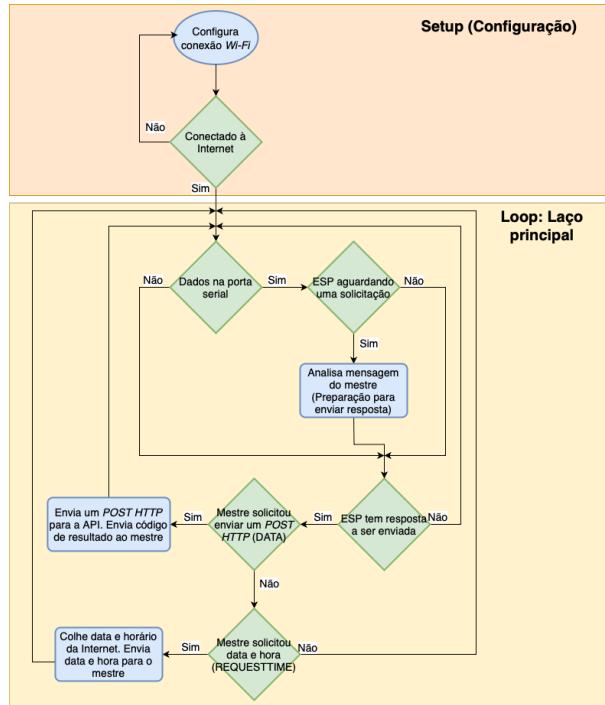
F.1 CONFIGURAÇÃO E CONEXÃO *WI-FI*

Conforme mostra o fluxograma da 33, as primeiras ações que o programa executa são aquelas relacionadas ao estabelecimento da conexão à Internet via rede *Wi-Fi* e a comunicação serial com o microcontrolador "mestre". Isso é realizado na função **setup()** conforme mostrado no código da Lista F.1.

Listing F.1 – Definição dos identificadores dos sensores de um dispositivo

```
void setup()
{
    Serial.begin(9600UL);
    Serial.print(F("+STARTESP;"));
    setup_wifi_connection<NUM_WIFIS>(wifiCreds);
    espHTTP.set_available(true);
    espSerial.set_status(WT_REQUEST);
    Serial.print(F("+ESPREADY;"));
}
```

Figura 33 – Fluxograma do firmware programado para o microcontrolador ESP8266



Fonte: Desenvolvido pelo autor (2023)

}

Primeiramente o programa inicializa a porta serial do ESP8266 a uma taxa de transmissão de 9600 bauds e imprime a mensagem '+STARTESP', indicando ao mestre que o programa foi inicializado. A função `setup_wifi_connection()` estabelece a conexão em uma rede *Wi-Fi* previamente armazenada na variável `wifiCreds`. Por fim, uma vez estabelecida a conexão, o objeto `espSerial` que armazena o estado da comunicação serial é setado para `WT_REQUEST` e a mensagem '+ESPREADY' é impressa, indicando que o ESP8266 está aguardando por uma solicitação do microcontrolador "mestre". A partir desse ponto o objeto `espHTTP` executa as requisições à API Renovar através do protocolo *HTTP*. A continuação são descritos os objetos, constantes e funções usados nesta parte do código.

F.1.1 NUM_WIFIS

Esta constante define a quantidade de redes *Wi-Fi* com as quais o ESP8266 tentará estabelecer uma conexão. Esta constante deve ser declarada antes do objeto `wifiCreds` e antes de invocar a função `setup_wifi_connection()`.

F.1.2 WiFiCredentials wifiCreds []

Esta variável é uma lista de objetos do tipo `WiFiCredentials`. Esta matriz armazena o SSID da rede, a senha e o nome de usuário (este último apenas em redes *WPA3 ENTERPRISE*). A lista `wifiCreds` deve ser declarada antes da função `setup()`,

conforme se mostra no código da Lista F.1.2. O tamanho da lista vai depender do valor da constante `NUM_WIFIS`.

Listing F.2 – Definição dos identificadores dos sensores de um dispositivo

```
/* Credenciais de uma rede empresarial WPA3*/
const WiFiCredentials CRED_1( "ssid1" , "senha1" , ENTERPRISE, "nomedeu

/* Credenciais das redes pessoais WPA3*/
const WiFiCredentials CRED_2( "ssid2" , "senha2" , PERSONAL);
const WiFiCredentials CRED_3( "ssid3" , "senha3" , PERSONAL);

/* Declara o array wifiCreds */
const WiFiCredentials wifiCreds [NUM_WIFIS] = { CRED_1, CRED_2, CRED_3}
```

F.1.3 `setup_wifi_connection<NUM_WIFIS>(wifiCreds)`

Esta função estabelece uma conexão a uma rede *Wi-Fi*. É definido como um *template* que recebe a quantidade de redes *Wi-Fi* armazenadas em `wifiCreds`.

F.1.4 `espHTTP`

Esta variável é um objeto da classe `HTTPHandler`, definida no arquivo `esp-iot.h`, cujo objetivo é encapsular as funcionalidades relacionadas às operações *HTTP*.

F.1.5 `espSerial`

Esta variável é um objeto da classe `ESPSerialHandler`, definida no arquivo `esp-serial-iot.h`. O objetivo deste objeto é encapsular as funcionalidades relacionadas à comunicação serial entre o ESP8266 e o microcontrolador mestre. Dependendo do estado deste objeto, o ESP8266 pode ler uma mensagem serial do mestre, ou executar uma determinada operação e enviar seu resultado de volta ao mestre. Para a versão atual do firmware, foram implementados dois estados:

`WT_REQUEST`: Indica que o ESP8266 não recebeu nenhuma solicitação do mestre e está aguardando até receber uma nova. `WT_RESPONSE`: Indica que o ESP8266 recebeu alguma nova solicitação do mestre e está realizando as operações para enviar uma resposta.

F.1.6 `Serial`

Este é um objeto do framework Arduino para controlar a comunicação serial.

F.2 O LAÇO PRINCIPAL

O laço principal do programa é executado dentro da função `loop()`, conforme mostrado no código abaixo. Esta função é responsável por monitorar a porta serial do ESP8266 e atender às solicitações do "mestre", conforme já ilustrado na Figura 33.

Listing F.3 – Laço principal do programa

```
void loop()
{
    static CommandTypes _cmdType = ERROR;

    if(Serial.available())
    {
        String serial_Str = Serial.readStringUntil(';' );
        if(espSerial.get_status() == WT_REQUEST)    _cmdType = espSerial.p
            Serial.flush();

        if(espSerial.get_status() == WT_RESPONSE)
        {
            espSerial.set_status(WT_REQUEST);
            switch (_cmdType)
            {
                case DATA:
                {
                    static uint8_t _numberOfPostTries = 0;
                    #define MAX_NUM_TRIES 3
                    int code = espHTTP.post(HOST, PORT, URL, espSerial.get_da

                    if(code <= 0 && WiFi.status() != WL_CONNECTED)
                    {
                        setup_wifi_connection<NUM_WIFIS>(wifiCreds);
                        if(++_numberOfPostTries >= MAX_NUM_TRIES-1) ESP.restart();
                    }
                    else    espSerial.send_http_code(code, Serial);
                    break;
                }

                case REQUESTTIME:
                {
                    static uint8_t _numberOfTries = 0;
```

```

#define MAX_NUM_TRIES 3
time_t t = get_time(TIMEZONE_SEC, DAYLIGHTOFFSET_SEC);
if (!t && WiFi.status() != WL_CONNECTED)
{
    setup_wifi_connection<NUM_WIFIS>(wifiCreds);
    if (++_numberOfTries >= MAX_NUM_TRIES - 1) ESP.restart();
}
else espSerial.send_time(t, Serial);
break;
}

default:
{
    break;
}
}
}
}

```

Verificação das solicitações do "mestre"

Sempre que houver dados na porta serial e o estado da comunicação serial (armazenado no objeto `espSerial`) for “aguardando solicitação” (`WT_REQUEST`), o mesmo objeto `espSerial` analisará a cadeia de caracteres recebida do “mestre” para determinar o tipo de solicitação que recebeu. Isso é feito dentro do primeiro `if` da função `loop()`, conforme mostrado no código da Lista F.2.

Listing F.4 – Código para verificar as solicitações do mestre

```

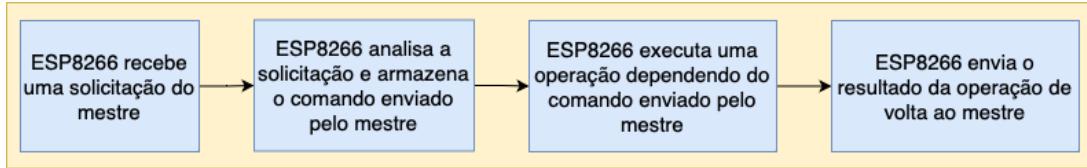
if (Serial.available())
{
    String serial_Str = Serial.readStringUntil(';' );
    if (espSerial.get_status() == WT_REQUEST) _cmdType = espSerial.parse(
        Serial.flush());
}

```

Uma vez analisada a mensagem do “mestre”, `espSerial` definirá seu estado para “aguardando resposta” (`WT_RESPONSE`), indicando que uma operação está em execução e uma resposta deve ser enviada de volta para o mestre (isso é feito internamente na função `parse`). Se o status de `espSerial` for `WT_RESPONSE`, o programa realizará uma operação `switch` para determinar qual operação o microcontrolador deve executar a seguir (fluxograma da Figura 33). A seleção da operação dependerá do tipo de solicitação enviado

pelo mestre. O tipo de solicitação é armazenado na variável `_cmdType` como resultado da operação `parse()` executada por `espSerial`. A Figura 34 mostra esse processo.

Figura 34 – Processo de atendimento a uma solicitação do mestre



Fonte: Desenvolvido pelo autor (2023)

Na versão atual do firmware, o ESP8266 aceita dois tipos de solicitações ou comandos do mestre: (1) uma solicitação para enviar uma requisição à API Renovar via um *POST HTTP* ou (2) uma solicitação de data e horário. A estrutura de uma mensagem de solicitação é mostrada no JSON da Lista F.2. Depois que o objeto `espSerial` processa a mensagem do mestre, ele armazena o tipo de solicitação na variável `_cmdType`. Esta variável é uma enumeração do tipo `CommandTypes` e, para a versão atual, pode conter os valores `ERROR`, `DATA` e `REQUESTTIME`, veja a Tabela 8.

Listing F.5 – O formato da string JSON trocada entre o ESP8266 e o mestre

```
{
    'type': // The type of the request. Could be 1 (DATA) or 2 (REQUESTTIME)
    'body': // The body of the request. Only used when the master is sending a request
}
```

Tabela 8 – Tipos de solicitações representadas no tipo `CommandTypes`

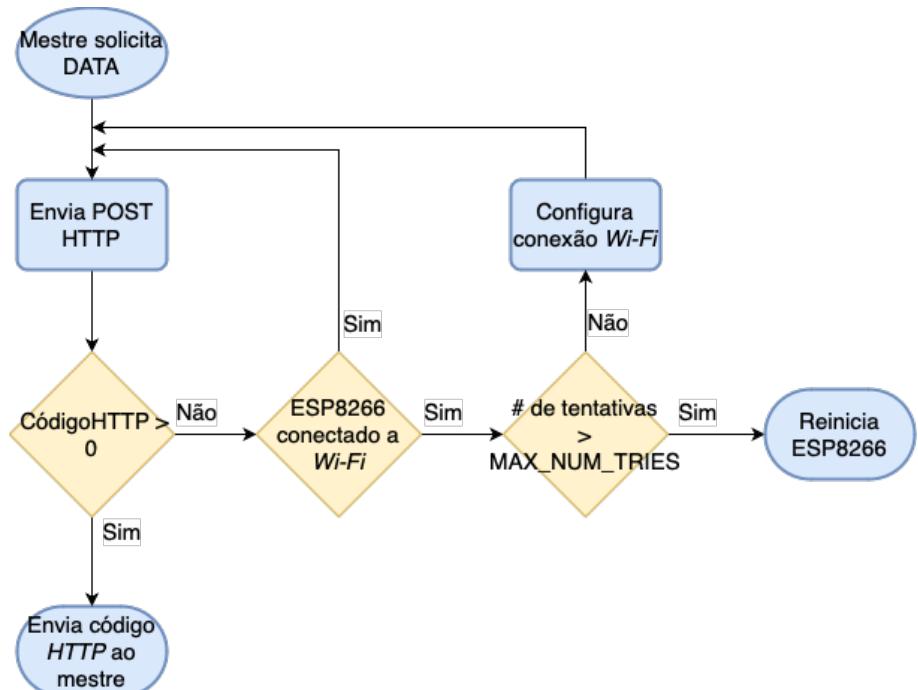
<code>CommandTypes</code>	Valor	Descrição	Resposta do mestre
<code>ERROR</code>	0	Este comando indica que ocorreu um erro na comunicação do mestre com o ESP8266. Normalmente usado do ESP8266 para o mestre	É a resposta em caso de erro de comunicação
<code>DATA</code>	1	Este comando indica que o mestre enviou os dados para uma postagem HTTP e está aguardando o código HTTP retornado do servidor Web como resposta à postagem	O código HTTP retornado pelo servidor Web após a postagem
<code>REQUESTTIME</code>	2	Este comando indica que o mestre está solicitando a hora de um servidor NTP	O carimbo de data/hora obtido pelo servidor NTP

Fonte: Desenvolvido pelo autor (2023)

F.2.1 O comando DATA: enviando um POST HTTP para a API Renovar

O código executado quando o "mestre" solicita o envio de um *POST HTTP* é mostrado na Lista F.2.1. O objeto `espHTTP` faz uma requisição tipo *POST* para a API hospedada em um servidor web identificado por um *HOST* (hospedeiro), uma Porta e uma URL específica. Os dados enviados no *POST* são os mesmos enviados anteriormente pelo mestre. Estes dados são acessados pelo método `get_data()` de `espSerial`. O código retornado dessa operação é enviado como resposta ao "mestre" para manter controle das suas operações. Caso o código for de falha e o ESP8266 não estiver conectado à rede *Wi-Fi*, o programa tentará se reconectar à rede e repassar os dados no máximo por três tentativas. Se, por outro lado, o código for de falha mas o ESP8266 estiver conectado a uma rede *Wi-Fi*, o programa tentará enviar a mesma requisição indefinidamente até que o mestre envie uma nova solicitação. A Figura 35 apresenta um fluxograma que representa esse processo.

Figura 35 – Fluxograma do processo após uma solicitação de DATA do mestre.



Fonte: Desenvolvido pelo autor (2023)

Listing F.6 – Sequencia de operações no comando DATA

```

case DATA:
{
    static uint8_t _numberOfPostTries = 0;
#define MAX_NUM_TRIES 3
    int code = espHTTP.post(HOST, PORT, URL, espSerial.get_data());
}

```

```

if( code <= 0 && WiFi. status () != WL_CONNECTED)
{
    setup_wifi_connection<NUM_WIFI>( wifiCreds );
    if(++_numberOfPostTries >= MAX_NUM_TRIES-1) ESP. restart ();
}
else    espSerial.send_http_code( code , Serial );
break;
}

```

Os valores de HOST, PORT e URL são definidos no arquivo iot-generic.h conforme se mostra abaixo; eles representam o *endpoint* para enviar requisições à API Renovar.

```

#define HOST  F( "renovar.lcqar.ufsc.br" )
#define PORT 8080UL
#define URL   F( "/sample/" )

```

O comando REQUESTTIME: obtendo a hora de um servidor NTP

O código executado quando o ESP8266 recebe uma solicitação de horário é mostrado na Lista F.2.1. A função `get_time()` é definida no arquivo `esp-iot.h` para obter a data e hora atuais desde um servidor NTP. O valor retornado dessa operação é enviado como resposta ao "mestre". Caso o valor for zero e o ESP8266 não esteja conectado à rede Wi-Fi, o programa tentará se reconectar à rede e obter o horário da Internet por no máximo três tentativas. Por outro lado, se o valor for zero, mas o ESP8266 estiver conectado a uma rede Wi-Fi, o programa tentará obter o horário da Internet indefinidamente até receber uma nova solicitação. A Figura 36 apresenta um fluxograma que representa esse processo.

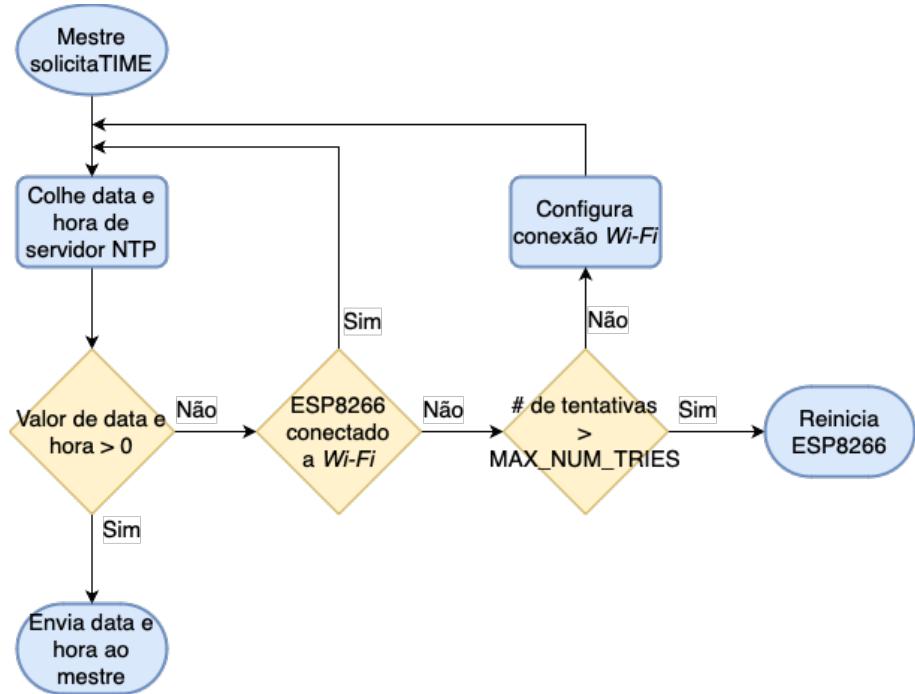
Listing F.7 – Sequencia de operações no comando REQUESTTIME

```

case REQUESTTIME:
{
    static uint8_t _numberOfTries = 0;
#define MAX_NUM_TRIES 3
    time_t t = get_time(TIMEZONE_SEC, DAYLIGHTOFFSET_SEC);
    if (!t && WiFi. status () != WL_CONNECTED)
    {
        setup_wifi_connection<NUM_WIFI>( wifiCreds );
        if(++_numberOfTries >= MAX_NUM_TRIES-1) ESP. restart ();
    }
    else    espSerial.send_time( t , Serial );
    break;
}

```

Figura 36 – Fluxograma do processo após uma solicitação TIME do mestre



Fonte: Desenvolvido pelo autor (2023)

A função `get_time()` recebe os parâmetros `TIMEZONE_SEC` e `DAYLIGHTOFFSET_SEC`, definidos anteriormente no arquivo `main.cpp`. O exemplo abaixo mostra como definir esses dois parâmetros para a aplicação no Brasil. `TIMEZONE_SEC` é o fuso horário onde o monitor será instalado, convertido a segundos. `DAYLIGHTOFFSET_SEC` define o deslocamento, em segundos, para o horário de verão, nesse caso foi definido como zero.

```
#define TIMEZONE_SEC      -3*3600
#define DAYLIGHTOFFSET_SEC 0*3600
```

ANEXO A – DOCUMENTAÇÃO DA API RENOVAR: *ENDPOINTS E REQUISIÇÕES*

Api Documentation

Api Documentation

Apache 2.0

basic-error-controller : Basic Error Controller

DELETE /error

errorHtml

Response Class (Status 200)

OK

Example Value

```
{  
    "empty": true,  
    "model": {},  
    "modelMap": {},  
    "reference": true,  
    "status": "100",  
    "view": {  
        "contentType": "string"  
    },  
    "viewName": "string"  
}
```

Response Content Type

Response Messages

HTTP Status Code	Reason	Response Model	Headers
204	No Content		
401	Unauthorized		
403	Forbidden		

GET /error

errorHtml

Response Class (Status 200)

OK

Example Value

```
{  
    "empty": true,  
    "model": {},  
    "modelMap": {},  
    "reference": true,  
    "status": "100",  
    "view": {  
        "contentType": "string"  
    },  
    "viewName": "string"  
}
```

Response Content Type [text/html](#)

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

HEAD /error errorHtml

Response Class (Status 200)

OK

Example Value

```
{  
    "empty": true,  
    "model": {},  
    "modelMap": {},  
    "reference": true,  
    "status": "100",  
    "view": {  
        "contentType": "string"  
    },  
    "viewName": "string"  
}
```

Response Content Type [text/html](#)

Response Messages

HTTP Status Code	Reason	Response Model	Headers
204	No Content		
401	Unauthorized		
403	Forbidden		

OPTIONS /error

errorHtml

Response Class (Status 200)

OK

Example Value

```
{  
  "empty": true,  
  "model": {},  
  "modelMap": {},  
  "reference": true,  
  "status": "100",  
  "view": {  
    "contentType": "string"  
  },  
  "viewName": "string"  
}
```

Response Content Type

Response Messages

HTTP Status Code	Reason	Response Model	Headers
204	No Content		
401	Unauthorized		
403	Forbidden		

PATCH /error

errorHtml

Response Class (Status 200)

OK

Example Value

```
{  
  "empty": true,  
  "model": {},  
  "modelMap": {},  
  "reference": true,  
  "status": "100",  
  "view": {  
    "contentType": "string"  
  },  
  "viewName": "string"  
}
```

Response Content Type

Response Messages

HTTP Status	Reason	Response Model	Headers
-------------	--------	----------------	---------

Code

204 No Content

401 Unauthorized

403 Forbidden

Try it out!

POST /error

errorHtml

Response Class (Status 200)

OK

Example Value

```
{  
  "empty": true,  
  "model": {},  
  "modelMap": {},  
  "reference": true,  
  "status": "100",  
  "view": {  
    "contentType": "string"  
  },  
  "viewName": "string"  
}
```

Response Content Type

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

PUT /error

errorHtml

Response Class (Status 200)

OK

Example Value

```
{
  "empty": true,
  "model": {},
  "modelMap": {},
  "reference": true,
  "status": "100",
  "view": {
    "contentType": "string"
  },
  "viewName": "string"
}
```

Response Content Type [text/html](#)

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

coordinate-resource : Coordinate Resource

GET	/coordinate	getAllCoordinates
-----	-------------	-------------------

Response Class (Status 200)

OK

Example Value

```
[
  {
    "id": 0,
    "latitude": 0,
    "longitude": 0
  }
]
```

Response Content Type [/*](#)

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		

403

Forbidden

404

Not Found

[Try it out!](#)

POST /coordinate

insertDevice

Parameters

Parameter	Value	Description	Parameter Type	Data Type
coordinate	<input type="text"/>	coordinate	body	Example Value <pre>{ "id": 0, "latitude": 0, "longitude": 0 }</pre>

Parameter content type:
`application/json`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /coordinate/check/{latitude}/{longitude}

getCoordinate

Response Class (Status 200)

boolean

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
latitude	<input type="text"/>	latitude	path	double
longitude	<input type="text"/>	longitude	path	double

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /coordinate/lat/{lat}

getCoordinatesByLatitude

Response Class (Status 200)

OK

Example Value

```
[  
  {  
    "id": 0,  
    "latitude": 0,  
    "longitude": 0  
  }  
]
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
latitude	<input type="text"/>	latitude	query	double

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /coordinate/long/{long}

getCoordinatesByLongitude

Response Class (Status 200)

OK

Example Value

```
[  
  {  
    "id": 0,  
    "latitude": 0,  
    "longitude": 0  
  }  
]
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
longitude	<input type="text"/>	longitude	query	double

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /coordinate/unique/{sensorID}

getUniqueCoordinateFromSamples

Response Class (Status 200)

OK

Example Value

```
[  
  {  
    "id": 0,  
    "latitude": 0,  
    "longitude": 0  
  }  
]
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
sensorID	<input type="text"/>	sensorID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

DELETE /coordinate/{id}

deleteDevice

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
204	No Content		
401	Unauthorized		
403	Forbidden		

[Try it out!](#)

GET /coordinate/{id}

getCoordinate

Response Class (Status 200)

OK

Example Value

```
{  
  "id": 0,  
  "latitude": 0,  
  "longitude": 0  
}
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

device-resource : Device Resource

GET `/device` [getAllDevices](#)

Response Class (Status 200)

OK

Example Value

```
[  
  {  
    "coordinate": {  
      "id": 0,  
      "latitude": 0,  
      "longitude": 0  
    },  
    "created_at": "2023-11-07T03:03:32.810Z",  
    "id": 0,  
    "name": "string",  
    "portable": true
```

Response Content Type `/*`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

POST /device insertDevice

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceDTO	<input type="text"/>	deviceDTO	body	<input type="text"/> Example Value

Parameter content type:

```
{
  "latitude": 0,
  "longitude": 0,
  "name": "string",
  "tracking": "string",
  "userID": 0
}
```

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /device/active/{deviceid} isDeviceActive

Response Class (Status 200)

boolean

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
-----------	-------	-------------	----------------	-----------

deviceid	<input type="text"/>	deviceid	path	integer
Response Messages				
HTTP Status Code	Reason	Response Model		Headers
401	Unauthorized			
403	Forbidden			
404	Not Found			
Try it out!				

GET	/device/coordid/{coordid}	getByCoordinateSample
Response Class (Status 200)		
OK		
Example Value		
<pre>[{ "coordinate": { "id": 0, "latitude": 0, "longitude": 0 }, "created_at": "2023-11-07T03:03:32.813Z", "id": 0, "name": "string", "portable": true</pre>		

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
coordid	<input type="text"/>	coordid	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		
Try it out!			

GET	/device/coordinate/{id}	getByCoordinate
-----	-------------------------	-----------------

Response Class (Status 200)

OK

Example Value

```
[  
 {  
   "latitude": 0,  
   "longitude": 0,  
   "name": "string",  
   "tracking": "string",  
   "userID": 0  
 }  
]
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /device/name/{name} [getDeviceByUser](#)

Response Class (Status 200)

OK

Example Value

```
[  
 {  
   "latitude": 0,  
   "longitude": 0,  
   "name": "string",  
   "tracking": "string",  
   "userID": 0  
 }  
]
```

Response Content Type `*/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
name	<input type="text"/>	name	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET [/device/nonportables](#) [getNonPortableDevice](#)

Response Class (Status 200)

OK

Example Value

```
[  
  {  
    "latitude": 0,  
    "longitude": 0,  
    "name": "string",  
    "tracking": "string",  
    "userID": 0  
  }  
]
```

Response Content Type `*/*`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET [/device/page](#) [buscarPagina](#)

Response Class (Status 200)

OK

Example Value

```
{
  "content": [
    {
      "latitude": 0,
      "longitude": 0,
      "name": "string",
      "tracking": "string",
      "userID": 0
    }
  ],
  "first": true
}
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
page	<input type="text"/>	page	query	integer
linesPerPage	<input type="text"/> 10	linesPerPage	query	integer
orderBy	<input type="text"/> id	orderBy	query	string
direction	<input type="text"/> ASC	direction	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /device/portables

getPortableDevice

Response Class (Status 200)

OK

Example Value

```
[
  {
    "latitude": 0,
    "longitude": 0,
    "name": "string",
    "tracking": "string",
    "userID": 0
  }
]
```

Response Content Type [/*](#)

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /device/user/{id} [getDeviceByUser](#)

Response Class (Status 200)

OK

Example Value

```
[
  {
    "latitude": 0,
    "longitude": 0,
    "name": "string",
    "tracking": "string",
    "userID": 0
  }
]
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
------------------	--------	----------------	---------

401 Unauthorized

403 Forbidden

404 Not Found

Try it out!

DELETE /device/{id}

deleteDevice

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
204	No Content		
401	Unauthorized		
403	Forbidden		

Try it out!

GET /device/{id}

getDevice

Response Class (Status 200)

OK

Example Value

```
{  
    "latitude": 0,  
    "longitude": 0,  
    "name": "string",  
    "tracking": "string",  
    "userID": 0  
}
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

PUT /device/{id}

updateDevice

Parameters

Parameter	Value	Description	Parameter Type	Data Type	Example Value
device	<input type="text"/>	device	body		<pre>{ "coordinate": { "id": 0, "latitude": 0, "longitude": 0 }, "created_at": "2023-11-07T03:03:32.822Z", "id": 0, "name": "string", "portable": true, "tracking_id": "string" }</pre>
id	<input type="text"/>	id	path	integer	

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

pollutant-resource : Pollutant Resource

GET	/pollutant		getAll
-----	------------	--	--------

Response Class (Status 200)

OK

Example Value

```
{}
```

Response Content Type `*/*`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

POST `/pollutant`

addPollutant

Parameters

Parameter	Value	Description	Parameter Type	Data Type	Example Value
<code>pollutantDTO</code>	<input type="text"/>	<code>pollutantDTO</code>	body		<pre>{ "max": 0, "name": "string", "unitId": 0 }</pre>

Parameter content type:

`application/json`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		

401 Unauthorized

403 Forbidden

404 Not Found

Try it out!

GET /pollutant/name/{name}

getPollutantByUnit

Response Class (Status 200)

OK

Example Value

```
{}
```

Response Content Type `*/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
name	<input type="text"/>	name	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /pollutant/sensor/{sensorID}

getPollutantBySensor

Response Class (Status 200)

OK

Example Value

```
{}
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
sensorID	<input type="text"/>	sensorID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

GET /pollutant/unit/{id} getPollutantByUnit

Response Class (Status 200)

OK

Example Value

```
{}
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
-----------	-------	-------------	----------------	-----------

id**id**

path

integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

DELETE /pollutant/{id} deletePollutant

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
204	No Content		
401	Unauthorized		
403	Forbidden		

[Try it out!](#)

GET /pollutant/{id} getPollutant

Response Class (Status 200)

OK

Example Value

```
{}
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text" value="1234567890"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

PUT /pollutant/{id} updatePollutant

Response Class (Status 200)

OK

Example Value

```
{}
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
pollutant	<input "co2",="" "value":="" 100}"="" name":="" type="text" value="{"/>	pollutant	body	<input type="button" value="Example Value"/>

Parameter content type:

```
{
  "id": 0,
  "max": 0,
  "name": "string",
  "unit": {
    "id": 0,
    "name": "string",
    "type": {
      "id": 0,
      "name": "string"
    }
  }
},
```

id	<input type="text"/>	id	path	integer
-----------	----------------------	-----------	------	---------

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

sample-resource : Sample Resource

GET	/sample	getAll
-----	---------	--------

Response Class (Status 200)

OK

Example Value

```
[
  {
    "coordinate": {
      "id": 0,
      "latitude": 0,
      "longitude": 0
    },
    "date": 0,
    "device": "string",
    "id": 0,
    "measuring": 0
  }
],
```

Response Content Type [/*](#)

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		

403

Forbidden

404

Not Found

[Try it out!](#)

POST /sample

Adds a sample

Parameters

Parameter	Value	Description	Parameter Type	Data Type
sampleDTO	<input type="text"/>	sampleDTO	body	Example Value <pre>{ "latitude": 0, "longitude": 0, "measuring": 0, "sensorid": 0, "timestamp": 0 }</pre>

Parameter content type:

 application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /sample/boxplot/

getBoxPlotValues

Response Class (Status 200)

OK

Example Value

```
[
  {
    "group": "string",
    "values": [
      0
    ]
  }
]
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
sensorID	<input type="text"/>	sensorID	query	integer
group	<input type="text"/>	group	query	string
start	<input type="text"/>	start	query	string
end	<input type="text"/>	end	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET </sample/coord/{id}> [getSamplesByCoordinate](#)

Response Class (Status 200)

OK

Example Value

```
[
  {
    "coordinate": {
      "id": 0,
      "latitude": 0,
      "longitude": 0
    },
    "date": 0,
    "device": "string",
    "id": 0,
    "measuring": 0
  }
]
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET </sample/device/active/{deviceID}> checkActiveDevice

Response Class (Status 200)

boolean

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceID	<input type="text"/>	deviceID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /sample/device/desc/{deviceID} getSamplesViaDeviceDESC

Response Class (Status 200)

OK

Example Value

```
[  
  {  
    "coordinate": {  
      "id": 0,  
      "latitude": 0,  
      "longitude": 0  
    },  
    "date": "2023-11-07T03:03:32.834Z",  
    "id": 0,  
    "latitude": 0,  
    "longitude": 0
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceID	<input type="text"/>	deviceID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /sample/device/last/{deviceID} getLastViaDevice

Response Class (Status 200)

OK

Example Value

```
{
  "content": [
    {
      "coordinate": {
        "id": 0,
        "latitude": 0,
        "longitude": 0
      },
      "date": "2023-11-07T03:03:32.835Z",
      "id": 0,
      "latitude": 0
    }
  ]
}
```

Response Content Type `*/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceID	<input type="text"/>	deviceID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET `/sample/device/pollutant/{deviceID}/{pollutantID}`

[getLastSampleViaPollutant](#)

Response Class (Status 200)

OK

Example Value

```
{
  "coordinate": {
    "id": 0,
    "latitude": 0,
    "longitude": 0
  },
  "date": "2023-11-07T03:03:32.836Z",
  "id": 0,
  "latitude": 0,
  "longitude": 0,
  "measuring": 0
}
```

Response Content Type `*/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
-----------	-------	-------------	----------------	-----------

deviceID	<input type="text"/>	deviceID	path	integer
pollutantID	<input type="text"/>	pollutantID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET	/sample/device/{deviceID}	getSamplesViaDevice
-----	---------------------------	---------------------

Response Class (Status 200)

OK

Example Value

```
[  
  {  
    "coordinate": {  
      "id": 0,  
      "latitude": 0,  
      "longitude": 0  
    },  
    "date": 0,  
    "device": "string",  
    "id": 0,  
    "measuring": 0
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceID	<input type="text"/>	deviceID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /sample/exists/{deviceID}/{pollutantID}

checkSampleExist

Response Class (Status 200)

OK

Example Value

```
{}
```

Response Content Type `*/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceID	<input type="text"/>	deviceID	path	integer
pollutantID	<input type="text"/>	pollutantID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /sample/page

searchPage

Response Class (Status 200)

OK

Example Value

```
{
  "content": [
    {
      "coordinate": {
        "id": 0,
        "latitude": 0,
        "longitude": 0
      },
      "date": "2023-11-07T03:03:32.840Z",
      "id": 0,
      "latitude": 0
    }
  ]
}
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceID	<input type="text"/>	deviceID	query	integer
page	<input type="text"/>	page	query	integer
linesPerPage	<input type="text"/> 5	linesPerPage	query	integer
orderBy	<input type="text"/> data	orderBy	query	string
direction	<input type="text"/> DESC	direction	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /sample/range/ [getPollutantDates](#)

Response Class (Status 200)

OK

Example Value

```
{
  "content": [
    {
      "coordinate": {
        "id": 0,
        "latitude": 0,
        "longitude": 0
      },
      "date": "2023-11-07T03:03:32.842Z",
      "id": 0,
      "latitude": 0
    }
  ]
}
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceID	<input type="text"/>	deviceID	query	integer
pollutantID	<input type="text"/>	pollutantID	query	integer
startDate	<input type="text"/>	startDate	query	string
endDate	<input type="text"/>	endDate	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET </sample/sensor/all/{id}> [getAllSamplesBySensor](#)

Response Class (Status 200)

OK

Example Value

```
[
  {
    "coordinate": {
      "id": 0,
      "latitude": 0,
      "longitude": 0
    },
    "date": "2023-11-07T03:03:32.844Z",
    "id": 0,
    "latitude": 0,
    "longitude": 0
  }
]
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET `/sample/sensor/last/{sensorID}`

[getLastSampleViaSensor](#)

Response Class (Status 200)

OK

[Example Value](#)

```
{
  "coordinate": {
    "id": 0,
    "latitude": 0,
    "longitude": 0
  },
  "date": "2023-11-07T03:03:32.845Z",
  "id": 0,
  "latitude": 0,
  "longitude": 0,
  "measuring": 0
}
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
-----------	-------	-------------	----------------	-----------

sensorID**sensorID**

path

integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

GET /sample/sensor/range/ getSensorDates

Response Class (Status 200)

OK

```
{
  "content": [
    {
      "coordinate": {
        "id": 0,
        "latitude": 0,
        "longitude": 0
      },
      "date": "2023-11-07T03:03:32.846Z",
      "id": 0,
      "latitude": 0
    }
  ]
}
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
sensorID	<input type="text"/>	sensorID	query	integer
startDate	<input type="text"/>	startDate	query	string
endDate	<input type="text"/>	endDate	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

GET /sample/sensor/{id} getSamplesBySensor

Response Class (Status 200)

OK

Example Value

```
{  
  "content": [  
    {  
      "coordinate": {  
        "id": 0,  
        "latitude": 0,  
        "longitude": 0  
      },  
      "date": "2023-11-07T03:03:32.847Z",  
      "id": 0,  
      "latitude": 0  
    }  
  ]  
}
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /sample/sensor/{sensorID}/device/{deviceID} getLastViaSensorDevice

Response Class (Status 200)

OK

Example Value

```
[
  {
    "coordinate": {
      "id": 0,
      "latitude": 0,
      "longitude": 0
    },
    "date": 0,
    "device": "string",
    "id": 0,
    "measuring": 0
  }
]
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
sensorID	<input type="text"/>	sensorID	path	integer
deviceID	<input type="text"/>	deviceID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET `/sample/{deviceID}/{pollutantID}`

getSamplesViaDevicePollutant

Response Class (Status 200)

OK

Example Value

```
[
  {
    "coordinate": {
      "id": 0,
      "latitude": 0,
      "longitude": 0
    },
    "date": "2023-11-07T03:03:32.849Z",
    "id": 0,
    "latitude": 0,
    "longitude": 0
  }
]
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceID	<input type="text"/>	deviceID	path	integer
pollutantID	<input type="text"/>	pollutantID	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET /sample/{id}

getSample

Response Class (Status 200)

OK

Example Value

```
{
  "coordinate": {
    "id": 0,
    "latitude": 0,
    "longitude": 0
  },
  "date": "2023-11-07T03:03:32.851Z",
  "id": 0,
  "latitude": 0,
  "longitude": 0,
  "measuring": 0
}
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

sensor-resource : Sensor Resource

GET /sensor

getAllSensors

Response Class (Status 200)

OK

Example Value

```
[  
 {  
   "brand": "string",  
   "created_at": "2023-11-07T03:03:32.852Z",  
   "device": {  
     "coordinate": {  
       "id": 0,  
       "latitude": 0,  
       "longitude": 0  
     },  
     "created_at": "2023-11-07T03:03:32.852Z"  
   }  
 }]
```

Response Content Type `/*`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

POST /sensor

addSensor

Response Class (Status 200)

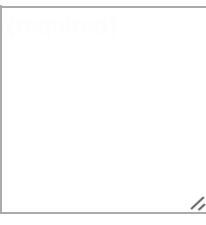
OK

Example Value

```
{  
   "brand": "string",  
   "created_at": "2023-11-07T03:03:32.853Z",  
   "device": {  
     "coordinate": {  
       "id": 0,  
       "latitude": 0,  
       "longitude": 0  
     },  
     "created_at": "2023-11-07T03:03:32.853Z",  
     "id": 0  
   }  
 }
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
sensorDTO		sensorDTO	body	Example Value <pre>{ "brand": "string", "deviceId": 0, "model": "string", "name": "string", "pollutantId": 0 }</pre>

Parameter content type:

`application/json`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET [/sensor/brand/{brand}](#) [getSensorsByBrand](#)

Response Class (Status 200)

OK

Example Value

```
[  
 {  
   "brand": "string",  
   "created_at": "2023-11-07T03:03:32.854Z",  
   "device": {  
     "coordinate": {  
       "id": 0,  
       "latitude": 0,  
       "longitude": 0  
     },  
     "created_at": "2023-11-07T03:03:32.851Z"  
   }  
 }]
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
-----------	-------	-------------	----------------	-----------

brand	<input type="text"/>	brand	path	string
Response Messages				
HTTP Status Code	Reason	Response Model		Headers
401	Unauthorized			
403	Forbidden			
404	Not Found			
Try it out!				

GET	/sensor/device/{id}	getSensorsByDevice
Response Class (Status 200)		
OK		
Example Value <pre>[{ "brand": "string", "created_at": "2023-11-07T03:03:32.855Z", "device": { "coordinate": { "id": 0, "latitude": 0, "longitude": 0 }, "created_at": "2023-11-07T03:03:32.855Z" } }]</pre>		

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		
Try it out!			

GET	/sensor/model/{model}	getSensorsByModel
-----	-----------------------	-----------------------------------

Response Class (Status 200)

OK

Example Value

```
[  
 {  
   "brand": "string",  
   "created_at": "2023-11-07T03:03:32.856Z",  
   "device": {  
     "coordinate": {  
       "id": 0,  
       "latitude": 0,  
       "longitude": 0  
     },  
     "created_at": "2023-11-07T03:03:32.856Z"  
   }  
 }]
```

Response Content Type

/*

Parameters

Parameter	Value	Description	Parameter Type	Data Type
model	<input type="text"/>	model	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /sensor/pollutant/{id} [getSensorsByPollutant](#)

Response Class (Status 200)

OK

Example Value

```
[  
 {  
   "brand": "string",  
   "created_at": "2023-11-07T03:03:32.856Z",  
   "device": {  
     "coordinate": {  
       "id": 0,  
       "latitude": 0,  
       "longitude": 0  
     },  
     "created_at": "2023-11-07T03:03:32.856Z"  
   }  
 }]
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET [/sensor/{id}](#) [getSensorsById](#)

Response Class (Status 200)

OK

Example Value

```
{  
  "brand": "string",  
  "created_at": "2023-11-07T03:03:32.857Z",  
  "device": {  
    "coordinate": {  
      "id": 0,  
      "latitude": 0,  
      "longitude": 0  
    },  
    "created_at": "2023-11-07T03:03:32.857Z",  
    "id": 0  
  }  
}
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

type-resources : Type Resources

GET /type

getAllTypes

Response Class (Status 200)

OK

Example Value

```
[  
  {  
    "id": 0,  
    "name": "string"  
  }  
]
```

Response Content Type `/*`

Response Messages

HTTP Status
Code

Reason

Response Model

Headers

401 Unauthorized

403 Forbidden

404 Not Found

Try it out!

POST /type

insert

Parameters

Parameter

Value

Description

Parameter
Type

Data Type

type

type

body

Example Value

Parameter content type:

application/json

```
{  
  "id": 0,  
  "name": "string"  
}
```

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /type/name/{name} getByName

Response Class (Status 200)

OK

Example Value

```
{  
  "id": 0,  
  "name": "string"  
}
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
name	<input type="text"/>	name	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET	/type/{id}	getById
-----	------------	---------

Response Class (Status 200)

OK

Example Value

```
{
  "id": 0,
  "name": "string"
}
```

Response Content Type [/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

unit-resources : Unit Resources

GET	/unit	getAllUnits
-----	-------	-------------

Response Class (Status 200)

OK

Example Value

```
[  
  {  
    "id": 0,  
    "name": "string",  
    "type": {  
      "id": 0,  
      "name": "string"  
    }  
  }  
]
```

Response Content Type `/*`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

POST /unit insert

Parameters

Parameter	Value	Description	Parameter Type	Data Type	Example Value
unitDTO	<input type="text"/>	unitDTO	body		<pre>{ "name": "string", "typeId": 0 }</pre>

Parameter content type:

`application/json`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		
401	Unauthorized		

403 Forbidden

404 Not Found

Try it out!

GET /unit/name/{name}

getByName

Response Class (Status 200)

OK

Example Value

```
{  
  "id": 0,  
  "name": "string",  
  "type": {  
    "id": 0,  
    "name": "string"  
  }  
}
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
name	<input type="text"/>	name	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /unit/type/{typeid}

getByType

Response Class (Status 200)

OK

Example Value

```
[
  {
    "id": 0,
    "name": "string",
    "type": {
      "id": 0,
      "name": "string"
    }
  }
]
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
typeid	<input type="text"/>	typeid	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET [/unit/{id}](#) [getUnit](#)

Response Class (Status 200)

OK

Example Value

```
{
  "id": 0,
  "name": "string",
  "type": {
    "id": 0,
    "name": "string"
  }
}
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
-----------	-------	-------------	----------------	-----------

id

[Required]

id

path

integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

user-resource : User Resource

GET	/user	getUsers
-----	-------	----------

Response Class (Status 200)

OK

[Example Value](#)

{}

[Response Content Type](#) [*/*](#)

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

POST /user

insertUser

Parameters

Parameter	Value	Description	Parameter Type	Data Type
-----------	-------	-------------	----------------	-----------

user	user	body	Example Value
			<pre>{ "created_at": "2023-11-07T03:03:32.868Z", "email": "string", "id": 0, "name": "string", "password": "string", "surname": "string" }</pre>

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

GET /user/page getPage

Response Class (Status 200)

OK

Example Value

```
{
  "content": [
    {
      "email": "string",
      "name": "string",
      "password": "string",
      "surname": "string"
    }
  ],
  "first": true,
  "last": true
}
```

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
page	<input type="text"/>	page	query	integer

linesPerPage	<input type="text" value="24"/>	linesPerPage	query	integer
orderBy	<input type="text" value="name"/>	orderBy	query	string
direction	<input type="text" value="ASC"/>	direction	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

GET	/user/{id}	getUser
-----	------------	---------

Response Class (Status 200)

OK

Example Value

```
{}
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)

PUT

/user/{id}

updateUser

Response Class (Status 200)

OK

Example Value

{}

Response Content Type `/*`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
user	<input type="text"/>	user	body	<h3>Example Value</h3> <pre>{ "created_at": "2023-11-07T03:03:32.870Z", "email": "string", "id": 0, "name": "string", "password": "string", "surname": "string" }</pre>
id	<input type="text"/>	id	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#)