

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA  
CATARINA – CÂMPUS FLORIANÓPOLIS  
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA  
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA**

**NYCOLAS COELHO DE ABREU**

**ESTUDO E DESENVOLVIMENTO DE PROGRAMAÇÕES LOW-CODE  
E NO-CODE PARA UM SISTEMA EMBARCADO**

**FLORIANÓPOLIS, 2022.**

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA  
CATARINA – CÂMPUS FLORIANÓPOLIS  
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA  
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA**

**NYCOLAS COELHO DE ABREU**

**ESTUDO E DESENVOLVIMENTO DE PROGRAMAÇÕES LOW-CODE  
E NO-CODE PARA UM SISTEMA EMBARCADO**

Trabalho de Conclusão de Curso  
submetido ao Instituto Federal de  
Educação, Ciência e Tecnologia de Santa  
Catarina como parte dos requisitos para  
obtenção do título de Engenheiro em  
Eletrônica.

Orientador:  
Prof. Renan Augusto Starke, Dr.

**FLORIANÓPOLIS, 2022**

Ficha de identificação da obra elaborada pelo autor.

Abreu, Nycolas Coelho de  
ESTUDO E DESENVOLVIMENTO DE PROGRAMAÇÕES LOW-CODE E  
NO-CODE PARA UM SISTEMA EMBARCADO  
/

Nycolas Coelho de Abreu; orientação de Renan  
Augusto Starke. - Florianópolis, SC, 2022.  
88 p.

Trabalho de Conclusão de Curso (TCC) - Instituto  
Federal de Santa Catarina, Câmpus Florianópolis.  
Bacharelado  
em Engenharia Eletrônica. Departamento  
Acadêmico de Eletrônica.  
Inclui Referências.

1. Programação em blocos.            2. Low-Code/No-Code.
3. Aplicação embarcada.   4. Ensino.
- I. Augusto Starke, Renan.
- II. Instituto Federal de Santa Catarina.
- III. ESTUDO E DESENVOLVIMENTO DE PROGRAMAÇÕES  
LOW-CODE E NO-CODE PARA UM SISTEMA EMBARCADO.

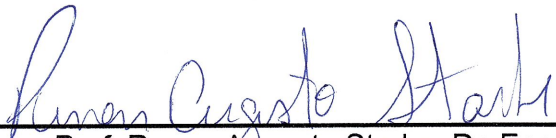
# **ESTUDO E DESENVOLVIMENTO DE PROGRAMAÇÕES LOW-CODE E NO-CODE PARA UM SISTEMA EMBARCADO**

**NYCOLAS COELHO DE ABREU**

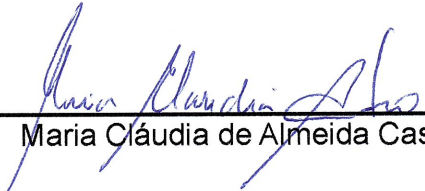
Este trabalho foi julgado adequado para obtenção do título de Engenheiro em Eletrônica e aprovado na sua forma final pela banca examinadora do Curso de Engenharia Eletrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 20 de dezembro, 2022.

Banca Examinadora:

  
Prof. Renan Augusto Starke, Dr. Eng.

  
Daniel Lohmann, Me. Eng.

  
Maria Cláudia de Almeida Castro, Dra. Eng.

## RESUMO

Diante do crescimento das ferramentas de programação *Low-Code* e *No-Code*, este trabalho apresenta a implementação e validação de uma aplicação embarcada, desenvolvida empregando linguagens de programação em blocos, identificando e elaborando os elementos de *hardware* e *software* cruciais para o controle de um kit de robótica educacional. Desta forma, habilitando estudantes e projetistas a realizar seus projetos e protótipos com soluções simples e modernas, podendo acelerar o período de aprendizado e entrega dos produtos ao mercado. Para tal propósito, realizou-se o projeto de uma placa de interface para o kit Buggy AWD (*All Wheel Drive*), sendo controlada pelo microcontrolador ESP 32. Para o controle deste kit utilizou-se como base de código a linguagem MicroPython, para definir os processos e periféricos mínimos a serem utilizados, desenvolvendo a partir desta base bibliotecas de programação em blocos aplicando as plataformas MicroBlocks e BIPES. Por último, são analisados e discutidos as limitações e funcionalidades de ambas as plataformas, viabilizando sua utilização.

**Palavras-chave:** Programação em blocos. Low-Code/No-Code. Aplicação embarcada. Ensino.

## **ABSTRACT**

Faced with the growth of Low-Code and No-Code programming tools, this work presents an implementation and validation of an embedded application, developed using block based programming languages. We also identified and elaborated crucial hardware and software elements to control an educational robotic kit using the ESP 32 microcontroller. Initially the MicroPython language was used as base specification in order to define some minimum processes to control all hardware peripherals. After that, the same functionality was developed using block platforms such as MicroBlocks and BIPES. Finally, we analyzed and discussed some limitations and functionalities of both blocks platforms.

**Keywords:** Block programming. Low-Code/No-Code. Embedded application. Teaching.

## LISTA DE FIGURAS

|                                                                                |    |
|--------------------------------------------------------------------------------|----|
| Figura 1 – Sintaxe, semântica e pragmática                                     | 19 |
| Figura 2 – Código compilado e interpretado                                     | 19 |
| Figura 3 – Caracteres de linguagens de programação                             | 20 |
| Figura 4 – Exemplo de código escrito em C embarcado                            | 22 |
| Figura 5 – PyBoard, microcontrolador oficial do MicroPython                    | 24 |
| Figura 6 – Ambiente de programação em blocos Scratch                           | 26 |
| Figura 7 – Diferença de performance de alunos em linguagem textual e em blocos | 28 |
| Figura 8 – Implementação da biblioteca Blockly                                 | 30 |
| Figura 9 – Interface do MicroBlocks                                            | 31 |
| Figura 10 – Interface da plataforma BIPES                                      | 35 |
| Figura 11 – Tipos de conexões do BIPES                                         | 36 |
| Figura 12 – Tradução do código em BIPES para MicroPython                       | 37 |
| Figura 13 – Buggy AWD                                                          | 38 |
| Figura 14 – Shield para MSP-EXP430G2                                           | 39 |
| Figura 15 – Circuito para monitoramento das baterias                           | 40 |
| Figura 16 – Sensor de linha TCRT5000                                           | 41 |
| Figura 17 – Circuito ponte H utilizado                                         | 42 |
| Figura 18 – ESP 32 Dev Kit v1                                                  | 43 |
| Figura 19 – Esquemático da placa de potência do Buggy AWD                      | 49 |
| Figura 20 – Pinos do ESP 32 Dev Kit v1                                         | 50 |
| Figura 21 – Esquemático da placa de interface                                  | 51 |
| Figura 22 – Layout da placa de interface                                       | 52 |
| Figura 23 – Modelo 3D da placa de interface                                    | 52 |
| Figura 24 – Diagrama UML de classes das classes de controle do Buggy AWD       | 54 |
| Figura 25 – Funcionamento da classe SensoresLinha                              | 55 |
| Figura 26 – Funcionamento da classe Motores                                    | 55 |
| Figura 27 – Funcionamento da classe Encoder                                    | 56 |
| Figura 28 – Funcionamento do sensor ultrassônico                               | 57 |
| Figura 29 – Funcionamento da classe Ultrassom                                  | 57 |
| Figura 30 – Funcionamento da classe Baterias                                   | 58 |
| Figura 31 – Processo de instalação do firmware do MicroBlocks                  | 59 |
| Figura 32 – Processo de criação de funções no MicroBlocks                      | 59 |

|                                                                                        |    |
|----------------------------------------------------------------------------------------|----|
| Figura 33 – Interface das funções desenvolvidas no MicroBlocks                         | 60 |
| Figura 34 – Definição das funções dos sensores de linha no MicroBlocks                 | 60 |
| Figura 35 – Definição das funções dos motores no MicroBlocks                           | 61 |
| Figura 36 – Exemplo de funcionamento e definição das funções do encoder no MicroBlocks | 62 |
| Figura 37 – Diagrama de tempo do funcionamento das funções do encoder no MicroBlocks   | 62 |
| Figura 38 – Definição da função do ultrassom no MicroBlocks                            | 63 |
| Figura 39 – Definição das funções de bateria no MicroBlocks                            | 64 |
| Figura 40 – Exportação da biblioteca no MicroBlocks                                    | 64 |
| Figura 41 – Plataforma de instalação <i>online</i> do MicroPython no BIPES             | 65 |
| Figura 42 – Processo de criação de funções no BIPES                                    | 66 |
| Figura 43 – Interface das funções desenvolvidas no BIPES                               | 66 |
| Figura 44 – Definição das funções dos sensores de linha no BIPES                       | 67 |
| Figura 45 – Definição das funções dos motores no BIPES                                 | 67 |
| Figura 46 – Definição das funções do encoder no BIPES                                  | 68 |
| Figura 47 – Definição das funções do ultrassom no BIPES                                | 68 |
| Figura 48 – Definição das funções de bateria no BIPES                                  | 69 |
| Figura 49 – Blocos para instalação e execução do Bluetooth                             | 69 |
| Figura 50 – Conexão com o terminal de console                                          | 70 |
| Figura 51 – Circuito para teste do funcionamento das bibliotecas                       | 71 |
| Figura 52 – Script seguidor de linha no MicroBlocks                                    | 71 |
| Figura 53 – Script seguidor de linha no BIPES                                          | 72 |

## **LISTA DE QUADROS**

|                                                |    |
|------------------------------------------------|----|
| Quadro 1 – Exemplos de símbolos                | 21 |
| Quadro 2 – Categorias de blocos do MicroBlocks | 32 |
| Quadro 3 – Placas suportadas pelo MicroBlocks  | 32 |
| Quadro 4 – Placas suportadas pelo BIPES        | 34 |
| Quadro 5 – Categorias de blocos do BIPES       | 35 |
| Quadro 6 – Módulos utilizados nas bibliotecas  | 44 |

## **LISTA DE TABELAS**

|                                                                                    |    |
|------------------------------------------------------------------------------------|----|
| Tabela 1 – Pinos e periféricos do ESP32-WROOM-32                                   | 43 |
| Tabela 2 – Características do MicroBlocks e BIPES                                  | 44 |
| Tabela 3 – Relação dos componentes e dispositivos adotados e sua respectiva função | 45 |
| Tabela 4 – Conexões com entre o ESP 32 e a placa de potência                       | 50 |
| Tabela 5 – Relação das classes de controle e suas respectivas responsabilidades    | 54 |
| Tabela 6 – Soluções implementadas para cada linguagem                              | 74 |

## **LISTA DE ABREVIATURAS E SIGLAS**

|                  |                                                      |
|------------------|------------------------------------------------------|
| ADC              | Analog-to-Digital Converter                          |
| ANSI             | American National Standards Institute                |
| API              | Application Programming Interface                    |
| AWD              | All Wheel Drive                                      |
| BIPES            | Block based Integrated Platform for Embedded Systems |
| CAN              | Controller Area Network                              |
| I/O              | Input / Output                                       |
| IDE              | Integrated Development Environment                   |
| IFRN             | Instituto Federal do Rio grande do Norte             |
| IFSC             | Instituto Federal de Santa Catarina                  |
| IoT              | Internet of Things                                   |
| I <sup>2</sup> C | Inter-Integrated Circuit                             |
| LED              | Light-Emitting Diode                                 |
| LiPo             | Lithium Polymer Battery                              |
| LPAE             | Laboratório de Pesquisa Avançada em Eletrônica       |
| MIT              | Massachusetts Institute of Technology                |
| PCB              | Printed Circuit Board                                |
| PID              | Proportional–Integral–Derivative                     |
| PWM              | Pulse Width Modulation                               |
| RAM              | Random Access Memory                                 |
| RPM              | Rotações Por Minuto                                  |
| SPI              | Serial Peripheral Interface                          |
| STL              | Standard Template Library                            |
| UART             | Universal Asynchronous Receiver-Transmitter          |

|        |                                             |
|--------|---------------------------------------------|
| UFRN   | Universidade Federal do Rio Grande do Norte |
| UFSCar | Universidade Federal de São Carlos          |
| USB    | Universal Serial Bus                        |
| USP    | Universidade de São Paulo                   |
| WDT    | Watchdog Timer                              |
| WiFi   | Wireless Fidelity                           |

## LISTA DE SÍMBOLOS

|      |                      |
|------|----------------------|
| A    | Amperes              |
| GHz  | Gigahertz            |
| KB   | Kilobytes            |
| Kb   | Kilobit              |
| MB   | Megabytes            |
| Mbps | Megabits por segundo |
| MHz  | Megahertz            |
| pA   | Pico Ampere          |
| V    | Volts                |

## SUMÁRIO

|                                                     |           |
|-----------------------------------------------------|-----------|
| <b>1 INTRODUÇÃO</b>                                 | <b>14</b> |
| 1.1 Justificativa                                   | 15        |
| 1.2 Definição do Problema                           | 16        |
| 1.3 Objetivo Geral                                  | 16        |
| 1.4 Objetivos Específicos                           | 17        |
| <b>2 FUNDAMENTAÇÃO TEÓRICA</b>                      | <b>18</b> |
| 2.1 Programação textual                             | 18        |
| 2.1.1 Sintaxe de linguagens textuais                | 20        |
| 2.1.2 Linguagem C                                   | 21        |
| 2.1.3 Linguagem C++                                 | 23        |
| 2.1.4 MicroPython                                   | 24        |
| 2.1.5 Futuro das linguagens de programação textuais | 25        |
| 2.2 Programação em blocos                           | 25        |
| 2.2.1 Ensino de programação em blocos               | 27        |
| 2.2.2 Scratch                                       | 28        |
| 2.2.3 Blockly                                       | 29        |
| 2.2.4 MicroBlocks                                   | 31        |
| 2.2.5 BIPES                                         | 34        |
| 2.3 Buggy AWD                                       | 37        |
| 2.3.1 Microcontrolador                              | 38        |
| 2.3.2 Sensores e módulos                            | 39        |
| 2.3.2.1 Encoder                                     | 39        |
| 2.3.2.2 Monitoramento de baterias                   | 40        |
| 2.3.2.3 Sensores seguidores de linha                | 40        |
| 2.3.2.4 Sensor de distância ultrassônico            | 41        |
| 2.3.3 Motores                                       | 41        |
| 2.4 ESP32                                           | 42        |
| 2.5 Resumo dos dispositivos e tecnologias adotados  | 44        |
| <b>3 METODOLOGIA</b>                                | <b>46</b> |
| <b>4 PROJETO E RESULTADOS</b>                       | <b>48</b> |
| 4.1 Placa de interface                              | 48        |
| 4.1.1 Placa de potência e de módulos do Buggy       | 48        |
| 4.1.2 Kit de desenvolvimento ESP 32 Dev Kit v1      | 49        |
| 4.1.3 Esquemático, layout e modelo 3D               | 51        |
| 4.2 Construção dos códigos                          | 53        |
| 4.2.1 MicroPython                                   | 53        |
| 4.2.2 MicroBlocks                                   | 58        |
| 4.2.3 BIPES                                         | 65        |

|                                                 |           |
|-------------------------------------------------|-----------|
| <b>4.3 Testes das bibliotecas</b>               | <b>70</b> |
| <b>4.4 Análise e discussão dos resultados</b>   | <b>73</b> |
| <b>5 CONSIDERAÇÕES FINAIS</b>                   | <b>76</b> |
| <b>REFERÊNCIAS</b>                              | <b>78</b> |
| <b>APÊNDICES</b>                                | <b>81</b> |
| APÊNDICE A – Classes de controle em MicroPython | 82        |

## 1 INTRODUÇÃO

O projeto de sistemas embarcados cresce de forma constante nos últimos anos. As aplicações são onipresentes na fabricação de produtos de consumo, veículos, telecomunicações, sistemas militares e de defesa do meio ambiente (MICHELI; SAMI, 1996).

Esses componentes estão se tornando mais complexos e possuem cada vez menos tempo para serem apresentados ao mercado. Portanto, surgiu a necessidade de criar um ambiente de projeto mais eficiente e de fácil utilização, onde os projetistas sem muita experiência sejam capazes de utilizar a programação visual ou *no-code* para auxiliar em seus estudos e protótipos.

Atualmente, a base da programação de microcontroladores são linguagens textuais como C, C++, Assembly e MicroPython, sendo todas linguagens com sintaxes rigorosas. Porém, como esclarece Caballar (2020), a programação *no-code* seria a progressão natural do desenvolvimento de *software*, envolvendo a adição de camadas de abstração e escondendo as complexidades por trás do código textual, objetivando uma programação mais fácil para os desenvolvedores, principalmente para iniciantes.

Ferramentas simples e intuitivas de programação *no-code*, como por exemplo a programação em blocos já são amplamente utilizadas por desenvolvedores e estudantes, plataformas de programação em blocos como o Scratch, que foi projetado para ensinar programação para jovens, possui mais de 95 milhões de usuários e quase 115 milhões de projetos compartilhados (MIT; 2022) e também a Blockly, uma biblioteca criada pelo Google para facilitar a criação de linguagens visuais em bloco, utilizada em centenas de projetos, principalmente educacionais (GOOGLE; 2022).

A proposta deste trabalho é avaliar a viabilidade das ferramentas de programação em blocos no desenvolvimento de sistemas embarcados, especificamente no controle de um carro autônomo AWD fornecido pelo LPAE (Laboratório de Pesquisa Avançada em Eletrônica). Para isso, será desenvolvida uma placa de interface de comunicação com o microcontrolador ESP 32, a fim de garantir a comunicação entre o sistema e o carro. Além disso, serão criadas

bibliotecas de programação utilizando tanto linguagem de programação textual quanto linguagens em blocos projetadas especificamente para microcontroladores.

A elaboração das bibliotecas foram realizadas pelas plataformas *open source* BIPES, desenvolvida no Brasil e planejada para programação de sistemas embarcados e aplicações IoT (*Internet of Things*) (JUNIOR et al., 2020), e também na plataforma MicroBlocks um ambiente de programação projetado para kits de eletrônica, inspirado em Scratch, GP e Snap! (CABRERA; MALONEY; WEINTROP, 2019).

## 1.1 Justificativa

A programação de microcontroladores é um processo que necessita de desenvolvedores e engenheiros qualificados com amplo conhecimento de uma determinada plataforma e linguagem de programação, além de requerer uma pesquisa minuciosa de diferentes documentos e *datasheets*, para que se consiga obter os conhecimentos necessários para a execução de um projeto. Como resultado, as bases de códigos geralmente carecem de modularidade e apresentam uma estrutura de difícil compreensão.

Para projetistas iniciantes, analisar e criar bases de código para microcontroladores torna-se uma tarefa difícil e demorada. Por esses motivos, novas ferramentas e metodologias de programação foram criadas para facilitar e acelerar o processo de aprendizado e prototipagem.

Um destes métodos é a programação em blocos, que introduz uma dimensão visual para apoiar o processo de aprendizagem. Segundo Weintrop (2019), este ambiente produz uma nova maneira de se escrever código, incluindo cores e formas, podendo reduzir a curva de aprendizado que os alunos possuem quando iniciam o aprendizado de programação em um ambiente baseado em texto.

A ferramenta de programação em blocos, além de facilitar a codificação para o usuário, prepara o ambiente de programação de forma automatizada, colocando em prática a ideia *plug-and-play*, o usuário não precisa se preocupar com diversas variáveis de preparação, instalação e configuração da plataforma.

Diante disso, este estudo tem como objetivo desenvolver uma interface para o microcontrolador ESP 32, a fim de abordar a programação em blocos voltada para microcontroladores, através da implementação de bibliotecas de código utilizando plataformas existentes e atuais de programação em blocos, realizando o controle de um kit de robótica educacional, com a finalidade de evidenciar a praticidade, viabilidade e funcionamento destas novas ferramentas de programação.

## 1.2 Definição do Problema

Dependendo da plataforma e da linguagem escolhida, a programação de sistemas embarcados pode ser complexa e demorada, envolvendo a compreensão de um conjunto de teoria, prática, conhecimento sobre paradigmas, sintaxe, semântica da linguagem, lógica e abstração. Esses aspectos dificultam o aprendizado para iniciantes e são uma barreira para aqueles que desejam criar protótipos rapidamente.

Assim, com as novas opções de linguagens e ferramentas de programação *Low-Code* e *No-Code*, podemos utilizá-las em conjunto a uma interface de *hardware* para o microcontrolador ESP 32 projetado para controle do kit de robótica educacional Buggy AWD, a fim de validar o funcionamento, limitações e melhorias de tais linguagens no processo de aprendizado de programação embarcada?

## 1.3 Objetivo Geral

O presente trabalho tem como objetivo o estudo e validação de ferramentas de programação alternativas e acessíveis, visando a facilidade na prototipação e uma alternativa de aprendizado moderna para programação de microcontroladores. Será desenvolvido um projeto de uma placa de interface para a plataforma de desenvolvimento ESP 32 Dev Kit v1, habilitando o controle do projeto do veículo autônomo AWD (STARKE; LPAE, 2022) desenvolvido pelo LPAE. Além disso, serão elaboradas bibliotecas de programação para o controle do veículo, utilizando MicroPython como base de código e as plataformas de programação em

blocos MicroBlocks e BIPES. As limitações e funcionalidades de cada plataforma serão avaliadas para determinar sua viabilidade.

#### **1.4 Objetivos Específicos**

Com o objetivo geral apresentado, destacam-se os objetivos específicos:

- a) Estudar as ferramentas e tecnologias disponíveis para o desenvolvimento das bibliotecas de código;
- b) Escolha dos componentes para construção da placa de interface;
- c) Projetar a interface de comunicação com o microcontrolador;
- d) Implementar as bibliotecas;
- e) Testar o funcionamento, viabilidade e usabilidade das tecnologias;
- f) Analisar os resultados alcançados;

## 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os tópicos principais relacionados ao trabalho. Inicialmente, é fornecido um embasamento teórico sobre programação textual e de blocos, enfatizando suas estruturas e características. Em seguida, são discutidas as linguagens de programação de blocos utilizadas neste projeto, destacando suas funcionalidades e particularidades. Por fim, é demonstrado o *hardware* e seus dispositivos que serão utilizados na elaboração deste projeto.

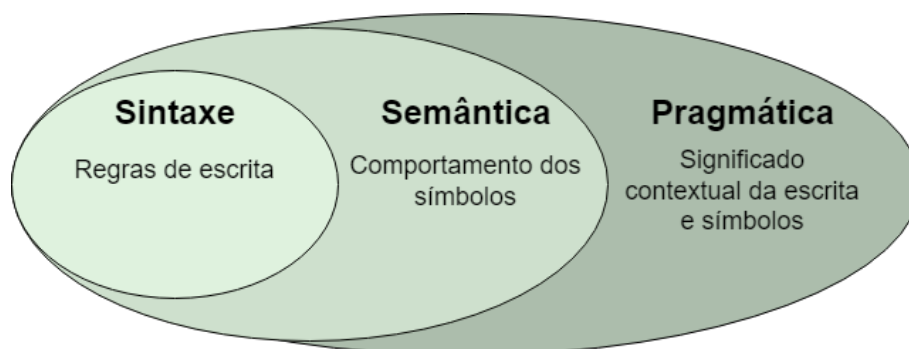
### 2.1 Programação textual

De acordo com Sammet (1969), uma linguagem de programação é um conjunto de caracteres que possuem regras para combiná-los e uma especificação de seus efeitos quando executados em um computador. O nível de abstração deve ser tal que os usuários não precisem conhecer código de máquina e para que a linguagem seja amplamente independente, permitindo que o mesmo programa seja executado em computadores com arquiteturas diferentes.

Slonneger e Kurtz (1995) definem as linguagens de programação textual pelas suas três características em comum:

- **Sintaxe:** Refere-se às formas como os símbolos podem ser combinados para criar comandos na linguagem. É uma descrição estrutural das expressões que compõem a linguagem. A sintaxe lida apenas com a forma e a estrutura dos símbolos, sem qualquer ligação com seus significados.
- **Semântica:** Revela o significado dos comandos em uma linguagem. Correlacionando seus símbolos com os comportamentos que devem ser executados.
- **Pragmática:** Trata-se dos aspectos da linguagem que envolvem os usuários, ou seja, fenômenos psicológicos e sociológicos como utilidade, escopo de aplicação e seus efeitos, incluindo questões como facilidade de implementação, eficiência na aplicação e metodologia de programação. A pragmática engloba todos os conceitos da sintaxe e semântica como apresentado na Figura 1.

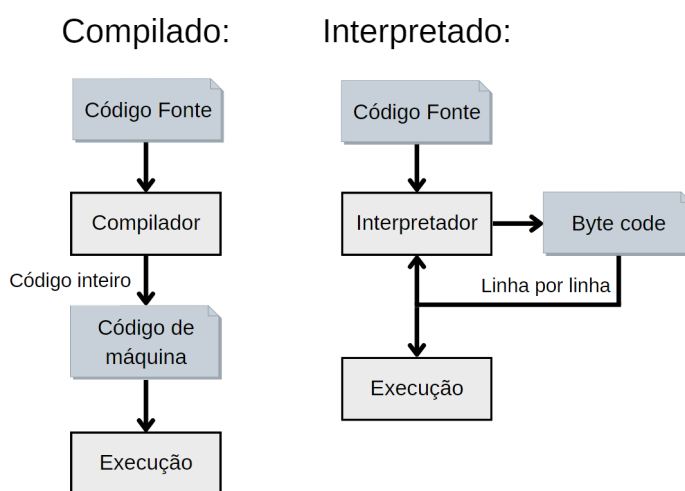
**Figura 1 – Sintaxe, semântica e pragmática**



Fonte: Elaboração própria (2022).

As linguagens de programação podem ser divididas em duas categorias, sendo elas linguagens compiladas e linguagens interpretadas. A Figura 2 ilustra como o código fonte é executado para ambos os tipos de linguagem.

**Figura 2 – Código compilado e interpretado**



Fonte: Elaboração própria (2022).

Em uma linguagem compilada, um compilador pré-traduz o programa em código de máquina para uma plataforma específica, podendo assim ser carregado na memória do computador e executado diretamente por ele (WATSON, 2017). Nas linguagens interpretadas não ocorre esta antecipação, o código é executado diretamente na plataforma pelo interpretador, traduzindo o programa em código de máquina linha por linha simultaneamente com a execução do programa (KWAME; MARTEY; CHRIS, 2017).

Os microcontroladores são tipicamente programados em linguagens compiladas como C e C++, contudo, como mostrado por Mazzei et al. (2015), com

desenvolvimento da tecnologia e o aumento da capacidade de memória dos microcontroladores, tornou-se possível implementar máquinas virtuais em ambientes embarcados, permitindo executar linguagens interpretadas e modernas como Java, Python e MicroPython.

### 2.1.1 Sintaxe de linguagens textuais

Um dos problemas das linguagens de programação é a diversidade de pessoas que precisam entender o seu funcionamento. Os usuários devem ser capazes de determinar como escrever soluções de software consultando o manual de referência, livros e outros documentos da linguagem. A sintaxe de uma linguagem de programação define as regras de escrita, delimitando a utilização dos símbolos e facilitando a detecção de erros (SEBESTA, 2018).

Segundo Horowitz (1984), toda linguagem de programação é baseada em um alfabeto de caracteres. Em inglês e português, o alfabeto possui 26 letras e mais 10 algarismos. Os conjuntos de caracteres para linguagens de programação geralmente começam com estes 36 caracteres mais a adição de caracteres especiais, como retratado pela Figura 3.

**Figura 3 – Caracteres de linguagens de programação**

|                                                                                   |
|-----------------------------------------------------------------------------------|
| <b>Letras</b><br>A B C D E F G H I J K L M N O P Q R S T U V W X Y Z              |
| <b>Algarismos</b><br>0 1 2 3 4 5 6 7 8 9                                          |
| <b>Caracteres especiais</b><br>= + - * / ( ) [ ] , . \$ ' : ! # % ^ &   \ ; " < > |

Fonte: Adaptado de Horowitz (1984).

Certos caracteres ou suas combinações que possuem significados bem definidos em um determinado contexto são chamados de símbolos, geralmente seus nomes ou lexemas, são reservados para que os desenvolvedores não os utilizem em outras situações. Símbolos reservados ajudam a tornar o código mais legível e

padronizado, mas à medida que o número de símbolos aumenta, torna-se mais difícil memorizá-los (HOROWITZ, 1984). O Quadro 1 exibe alguns dos símbolos e lexemas comumente utilizados por linguagens modernas.

**Quadro 1 – Exemplos de símbolos**

| <b>Símbolos</b> | <b>Lexemas</b>             |
|-----------------|----------------------------|
| Palavras-chave  | if, while, for, int, float |
| Separadores     | { } ( ) ; : [ ]            |
| Operadores      | / * - + = < >              |
| Identificadores | x, nota, pi, D2            |
| Literais        | true, "musica", 100        |

Fonte: Adaptado de Aho et al. (2007).

Como explica Aho et al. (2007), espera-se que os compiladores e interpretadores ajudem o programador a localizar e rastrear estes erros de sintaxe presentes nos programas:

- Erros lexicais: Incluem-se erros ortográficos de identificadores, palavras-chave ou operadores.
- Erros sintáticos: Ponto-e-vírgula mal posicionado ou chaves extras ou ausentes.
- Erros semânticos: Incompatibilidades de tipo entre operadores e operandos.
- Erros lógicos: Utilização de operadores de maneira indevida.

Linguagens de programação como C e C++ possuem uma sintaxe mais rigorosa, com a utilização de separadores para finalizar linhas e delimitar escopos, além do uso de palavras-chave para declaração de variáveis (AHO et al., 2007). Python por ser uma linguagem de alto nível, delimita seus escopos e linhas apenas com indentação, não necessitando de palavras-chave para declarar previamente uma variável.

### 2.1.2 Linguagem C

Como o criador da linguagem Ritchie (1993) afirma, C surgiu entre os anos de 1969 e 1973 como uma linguagem para ser utilizada no sistema operacional Unix, mas apenas em meados de 1980 a linguagem foi padronizada pelo comitê

ANSI (*American National Standards Institute*), fazendo seu uso se espalhar e colocando-a entre as linguagens mais usadas em toda a indústria de computadores.

C é uma linguagem de programação compilada, de uso geral, com paradigmas estruturais, imperativos e procedurais, possui uma sintaxe reduzida, controle de fluxo moderno, estruturas de dados e um rico conjunto de operadores. Sua ausência de limitações e sua generalidade a tornam mais conveniente e eficiente para muitas tarefas do que linguagens supostamente mais poderosas (KERNIGHAN; RITCHIE, 1988). Um exemplo de código C embarcado pode ser visualizado na Figura 4.

**Figura 4 – Exemplo de código escrito em C embarcado**

```
1. #include <msp430.h>
2.
3. int main(void)
4. {
5.     WDTCTL = WDTPW | WDTHOLD; // Para o watchdog timer
6.     P1DIR |= BIT0 | BIT1;      // Pino P1.0 e P1.1 como saída
7.
8.     while(1) {
9.         P1OUT |= BIT0 | BIT1; // Liga LED 1 e 2 SET_BIT
10.        __delay_cycles(5000); // Aguarda 5000 ciclos
11.        P1OUT&=~(BIT0|BIT1); // Desliga LED CLR_BIT
12.        __delay_cycles(5000); // Aguarda 5000 ciclos
13.    }
14.    return 0;
15. }
```

Fonte: Elaboração própria (2022).

Os microcontroladores são tradicionalmente programados em linguagem Assembly, como Pardue (2015) afirma, cada plataforma possui sua própria sintaxe, dificultando o aprendizado e tornando o código obsoleto no momento em que se muda de família. Alunos, professores e hobistas tendem a pular o estudo detalhado de Assembly, seguindo direto para o C, que é mais simples e eficiente. Devido a quantidade crescente de compiladores e sua facilidade de implementação, C se tornou a linguagem padrão para programação em microcontroladores (BATES, 2008).

Todos os microcontroladores possuem periféricos e outros componentes que devem ser acessados e controlados pelo programador. A linguagem C permite configurar e inicializar os bits sinalizadores nos registradores de controle para cada

operação, além de facilitar o acesso aos endereços e a memória interna do dispositivo, dando ao programador total controle da plataforma que está sendo utilizada.

### 2.1.3 Linguagem C++

Como Stroustrup (2013) o criador da linguagem explica, C++ foi desenvolvida em 1983, projetada para tornar a programação mais eficiente e moderna. C++ é uma extensão da linguagem de programação C, dessa forma ela possui retrocompatibilidade com os códigos já existentes escritos em C.

C++ é uma linguagem de programação compilada, de propósito geral, assim como C, mas com paradigmas modernos, incluindo abstração de dados, orientação a objetos e programação genérica. Ao longo do tempo, a linguagem passou por diversas atualizações, sendo a principal adição uma nova biblioteca padrão chamada STL (*Standard Template Library*) em 1998, que forneceu ferramentas úteis como contêineres, algoritmos e iteradores de contêineres (STROUSTRUP, 2013).

Entre as maiores diferenças com C inclui-se a programação orientada a objetos, que reduz o tempo de desenvolvimento, devido a facilidade de reutilização de código, também inclui-se a alocação dinâmica de memória utilizando ponteiros inteligentes, que diminui drasticamente a chance de falhas de segurança, estes pontos fazem a linguagem C++ ser mais popular que C em projetos de grande porte (BARR, 1999).

C++ foi projetado para programação de sistemas e *software* embarcado, focado em grandes sistemas e com recursos limitados, com desempenho, eficiência e flexibilidade de uso como destaques do design (STROUSTRUP, 2014). Portanto, devido a todos os fatores, a utilização de C++ em microcontroladores se torna uma experiência mais moderna e completa quando se comparada com a linguagem C.

#### 2.1.4 MicroPython

“Python é uma linguagem de script de alto nível, interpretada e orientada a objetos. Sua principal característica é a sua sintaxe clara e simples, é próximo do inglês, sendo possível ler o código e entendê-lo sem ter aprendido Python” (BEEL, 2017, p.19). A linguagem é utilizada no desenvolvimento de sites, aplicativos gráficos, análise de dados, aprendizado de máquina e IoT. Tornou-se popular devido a sua simplicidade e pelo seu conjunto de bibliotecas e API's (*Application Programming Interface*) desenvolvidas pela comunidade.

O MicroPython surgiu em 2013, como uma implementação do Python 3, desenvolvido por Damien George com foco em microcontroladores e sistemas embarcados, podendo executar códigos escritos em Python diretamente no hardware (TOLLERVEY, 2017). Ele apresenta um subconjunto da biblioteca padrão do Python, é otimizado para ambientes embarcados e compacto o suficiente para ser executado em apenas 256 kb de *flash* e 16 kb de RAM (*Random Access Memory*) (GEORGE, 2014). Na Figura 5 é exibido o microcontrolador oficial do MicroPython.

**Figura 5 – PyBoard, microcontrolador oficial do MicroPython**



Fonte: George (2014).

A linguagem possui diversos recursos para facilitar o desenvolvimento em projetos embarcados, incluindo bibliotecas de I/O (*Input/Output*), ADC (*Analog-to-Digital Converter*), PWM (*Pulse Width Modulation*), UART (*Universal Asynchronous Receiver-Transmitter*), SPI (*Serial Peripheral Interface*), I<sup>2</sup>C (*Inter-Integrated Circuit*), rede e *Bluetooth*. “Em geral, pode-se esperar que o

MicroPython suporte cerca de 80% das bibliotecas mais utilizadas em Python” (BEEL, 2017, p.30).

#### 2.1.5 Futuro das linguagens de programação textuais

A evolução das linguagens de programação tem sido uma constante em toda a história da computação. Desde a primeira linguagem de programação, a Assembly, até as linguagens de programação textuais mais populares, como Python, Java e C++, a necessidade de se desenvolver soluções mais eficientes e fáceis de usar levou à criação de novos tipos de linguagens de programação.

Entre essas novas linguagens, destacam-se as chamadas linguagens *Low-Code* e *No-Code*. Elas se diferem das linguagens de programação textuais tradicionais, pois são baseadas em programação por blocos e têm como objetivo simplificar a criação de soluções, são ideais para desenvolvedores menos experientes ou para profissionais que precisam criar soluções rapidamente, sem se preocupar com questões técnicas complexas (CABALLAR, 2020).

A programação em blocos tem se tornado cada vez mais popular, especialmente no mercado de educação, pois permite que crianças e adolescentes aprendam programação de forma lúdica e intuitiva. Além disso, é uma ótima forma de iniciar o aprendizado de programação, uma vez que o usuário pode visualizar o funcionamento do seu código em tempo real (WEINTROP; WILENSKY, 2015).

## 2.2 Programação em blocos

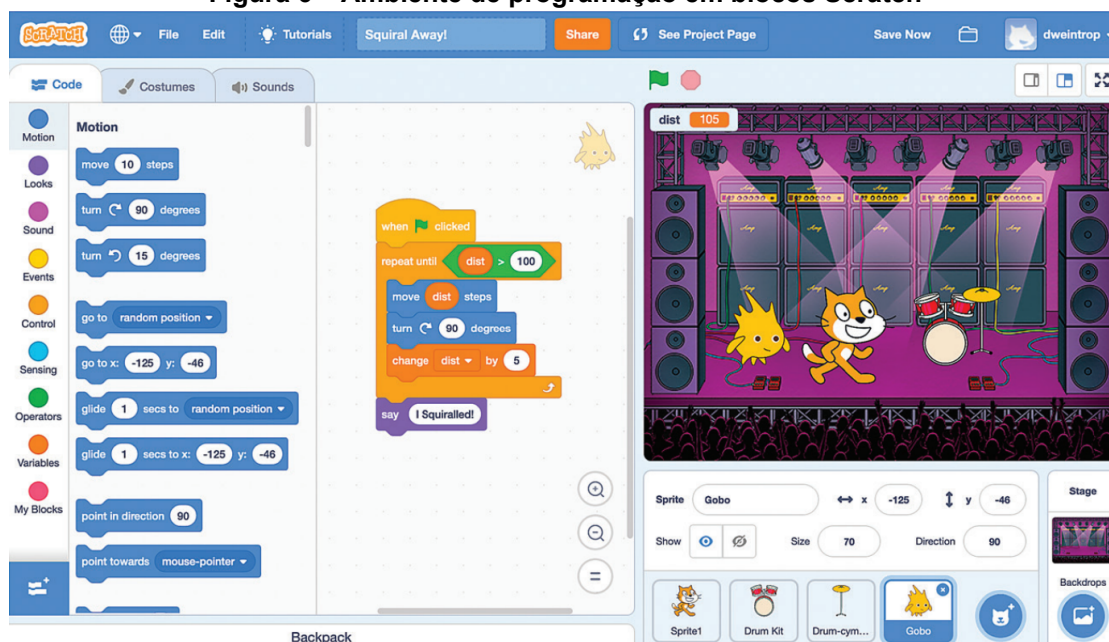
Como aponta Bau et al. (2017), as plataformas de programação baseadas em blocos utilizam metáforas de programação simples, como peças de um quebra-cabeça, para fornecer aos usuários dicas visuais sobre como e onde usar comandos para restringir a composição do programa. Permitindo que programadores e alunos concentrem-se na compreensão dos conceitos básicos de programação e na estrutura geral de um programa (MOORS; LUXTON-REILLY; DENNY, 2018).

A programação nesses ambientes assume a forma de arrastar blocos para a área de *script* e os agrupar para formar um comando. Se dois blocos não puderem ser unidos para formar uma instrução de sintaxe válida, o ambiente impede que eles sejam agrupados, evitando assim erros de sintaxe (MOORS; LUXTON-REILLY; DENNY, 2018).

A eliminação de erros de sintaxe torna-se uma das principais características dos ambientes de programação em blocos, pois iniciantes são capazes de se concentrar no fluxo de controle, condicionais e *loops*, sem a frustração de garantir que um programa seja sintaticamente correto (WEINTROP, 2019).

Ao contrário das linguagens textuais, as linguagens de bloco estão profundamente relacionadas ao seu ambiente de programação, ferramentas como *Scratch* e *Blockly* empregam algumas convenções de interface, como lidar com a organização, separando os blocos de comandos em abas relacionadas por categoria e adicionando cores diferentes para cada tipo de instrução (BAU et al., 2017), pode-se ver a combinação destes elementos na Figura 6.

**Figura 6 – Ambiente de programação em blocos Scratch**



Fonte: Weintrop (2019).

De acordo com Weintrop e Wilensky (2015), a programação baseada em blocos é conceituada por alguns como uma introdução à programação facilitando

uma eventual transição para a programação baseada em texto, sendo uma questão empírica aberta, que está apenas começando a ser respondida.

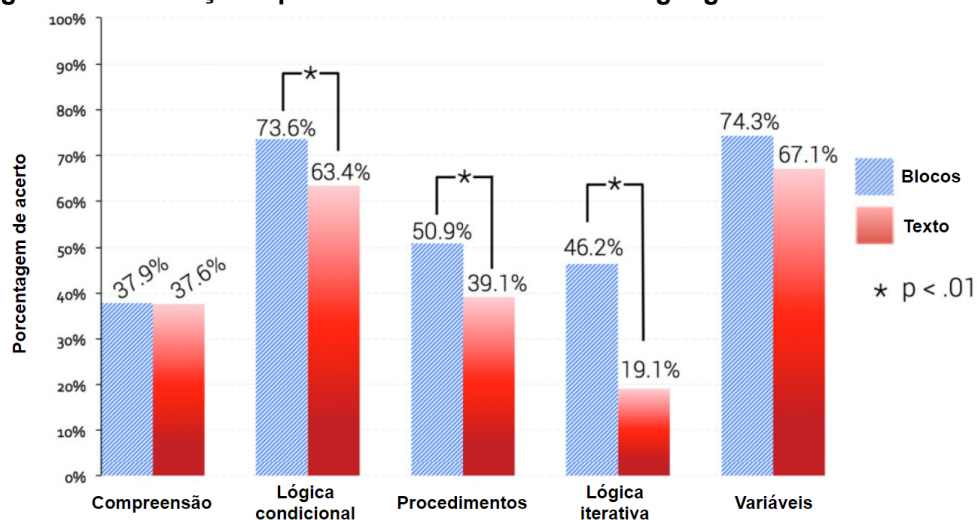
### 2.2.1 Ensino de programação em blocos

Algumas das vantagens da programação em blocos podem ocasionar problemas ao escalar o desenvolvimento para grandes projetos e equipes. Segundo Bau et al. (2017) os blocos ocupam mais espaço na tela, dificultando a navegação e a edição de seus comandos, além disso, pequenas edições em comandos existentes são mais difíceis, pois seus blocos precisam ser selecionados e excluídos individualmente.

Apesar de seus problemas, existe uma questão maior sobre o papel das ferramentas baseadas em blocos no campo da ciência da computação. Estas ferramentas servem como uma introdução para o ambiente de programação, em suma, para que uma ferramenta de programação possa ser usada por programadores novos ou casuais, seu *design* deve focar na capacidade de aprendizado, com a expectativa de que os alunos migrem para linguagens de programação textuais no futuro (WEINTROP, 2019).

Estudos comprovam que as linguagens de programação em blocos facilitam o aprendizado em diversos elementos da área de programação. Em uma pesquisa comutativa feita por Weintrop e Wilensky (2015), é demonstrado um estudo de 10 semanas com 90 alunos, entre a oitava série e terceiro ano, em um curso introdutório de programação.

Os alunos foram introduzidos durante 5 semanas a uma linguagem de programação semelhante ao Scratch, com atividades introdutórias e intermediárias. Sendo aplicado uma prova de 28 questões ao final deste período, a prova foi dividida entre os temas que foram apresentados. Ao fim da etapa de prova, os mesmos alunos participaram novamente durante 5 semanas em aulas introdutórias da linguagem textual Java, por meio de atividades similares às do período anterior. Ao encerrar este período, foi aplicado novamente uma prova de 28 questões, finalizando o estudo. A Figura 7 apresenta a porcentagem de acerto em média para cada categoria avaliada e sua respectiva linguagem utilizada.

**Figura 7 – Diferença de performance de alunos em linguagem textual e em blocos**

Fonte: Adaptado de Weintrop e Wilensky (2015).

Weintrop e Wilensky (2015) concluem que para todas as categorias apresentadas, houve uma melhoria no desempenho dos alunos quando foi utilizada a linguagem baseada em blocos comparando com a linguagem textual, como pode ser visto pelas barras de porcentagem na Figura 6, comprovando a ideia de que a linguagem de programação em blocos influencia positivamente no aprendizado de conceitos de programação para jovens.

Conhecer as vantagens e as desvantagens dos ambientes de programação baseados em blocos ajuda na compreensão de como introduzir estes ambientes para iniciantes. A ideia não é substituir a programação textual em disciplinas introdutórias ou na base de aprendizado, mas sim encorajar estes indivíduos e minimizar a lacuna entre linguagens baseadas em blocos e linguagens baseadas em texto, ampliando o conhecimento com novas tecnologias que facilitem o entendimento dos paradigmas de programação (WEINTROP, 2019).

### 2.2.2 Scratch

Scratch é uma linguagem de programação visual exclusiva para computadores, desenvolvida desde 2003 pelo MIT (*Massachusetts Institute of Technology*), financiado pela *National Science Foundation*, *Intel Foundation*, Nokia e pelo MIT *Media Lab Research Consortium*, coordenado por Mitchel Resnick e inspirado nas linguagens Logo e Squeak. “É a maior comunidade de programação

do mundo para crianças e uma linguagem de programação com uma interface visual simples que permite que os jovens criem histórias, jogos e animações digitais” (MIT, 2022).

Foi apresentado ao mundo pela primeira vez em 15 de maio de 2007, atualmente é usado em mais de 150 países e disponível em mais de 40 idiomas. Foi concebido para ajudar jovens a partir dos 8 anos. A ferramenta promove o pensamento computacional e desenvolve habilidades de resolução de problemas, auto-expressão, colaboração, lógica e programação (MIT, 2022).

Conforme apresentado pela Figura 6, o ambiente de desenvolvimento é dividido em três áreas. A primeira área contém os blocos de comando que serão adicionados ao programa, existem comandos relacionados ao movimento, fala, eventos e controle da área de animação. A segunda área é o desenvolvimento do programa, onde é possível adicionar, editar e remover os blocos de comando. A terceira área é a tela de animação, chamada de palco, onde o programa é executado e animado.

Um dos principais recursos do Scratch é seu *feedback* instantâneo, sem etapas de compilação ou distinções de modo de edição/execução. Os usuários podem clicar em um comando a qualquer momento para testar seu funcionamento. É possível alterar parâmetros ou adicionar blocos ao código enquanto o programa está em execução. Ao eliminar opções de modo potencialmente irritantes e pausas de construção, o Scratch ajuda os usuários a permanecerem envolvidos em testes, depuração e aprimoramento de seus projetos (MALONEY et al., 2010).

### 2.2.3 Blockly

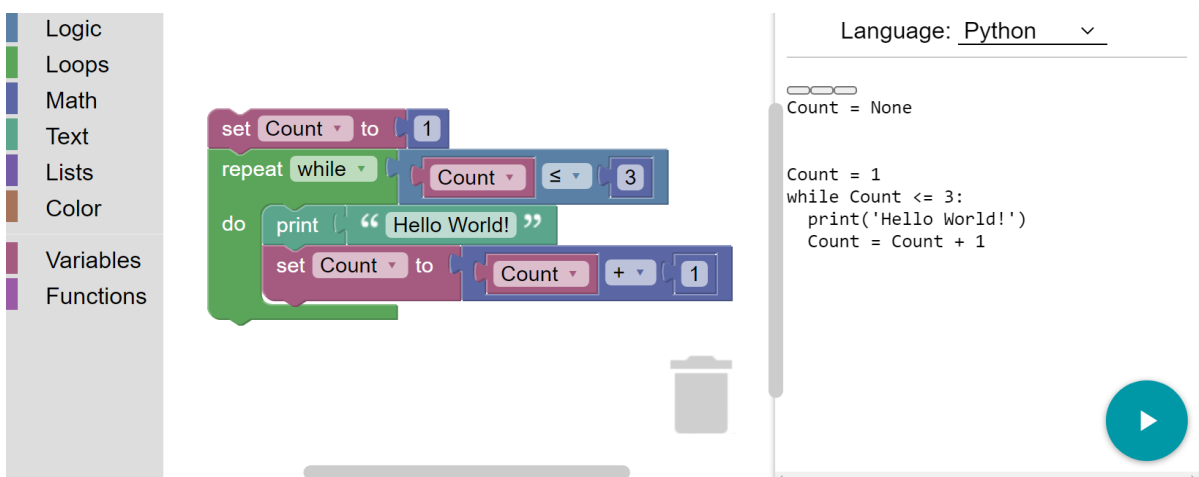
Blockly é uma biblioteca desenvolvida pelo Google em 2012 em colaboração com uma equipe de desenvolvimento do MIT. É utilizada para construir editores de programação em blocos para os mais diversos projetos. A biblioteca é gratuita e de código aberto, desenvolvida em Javascript, por ser uma ferramenta *web*, não requer instalação de nenhum recurso além de um navegador para funcionar corretamente (GOOGLE, 2022).

Não é uma linguagem de programação nem uma plataforma para usuários finais, constitui-se de uma API que facilita o desenvolvimento e a implementação de linguagens de programação em blocos. Fornece os recursos necessários para estruturar uma linguagem de programação em blocos, como uma interface gráfica para o editor de blocos, lógicas de conexões entre os blocos e um gerador de código para a linguagem textual.

Sua principal característica é a capacidade de exportar nativamente suas linguagens de bloco para linguagens de programação, incluindo JavaScript, Python, PHP, Lua e Dart. Caso seja necessária a tradução para outras linguagens, a biblioteca Blockly fornece todas as ferramentas para realizar esta tarefa. O processo de tradução da linguagem em blocos facilita o aprendizado do usuário e a adaptação às linguagens de programação textual (GOOGLE, 2022).

A Figura 8 mostra um exemplo de implementação da biblioteca Blockly. À esquerda, pode-se observar os blocos separados por categorias, blocos de lógica, loop, operações matemáticas, texto, listas, cores, variáveis e funções. No centro da imagem está a área de edição dos blocos. À direita, o código em blocos é transformado em linguagem de programação textual e pode ser executado apertando o botão de *play*.

**Figura 8 – Implementação da biblioteca Blockly**



Fonte: Google (2022).

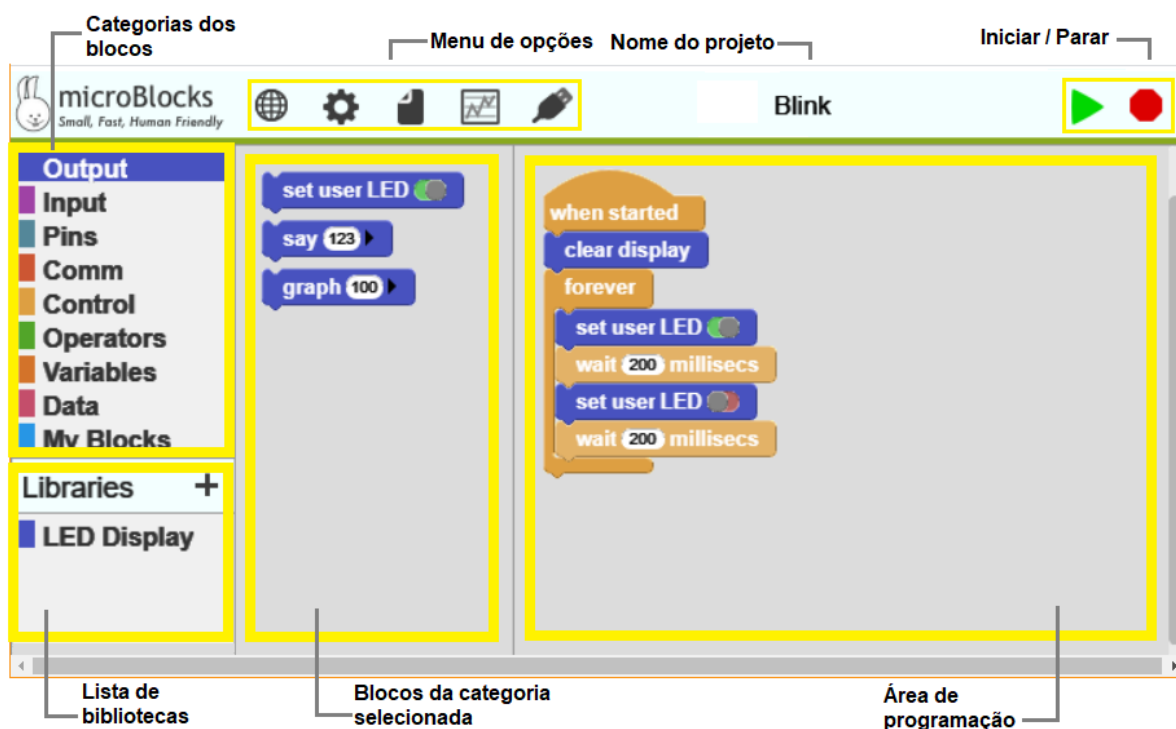
## 2.2.4 MicroBlocks

O MicroBlocks é um ambiente de linguagem de programação visual semelhante ao Scratch, focado no ensino de programação para microcontroladores educacionais, como o micro:bit, Adafruit Circuit Playground Express e muitos outros. Fundado por John Maloney e lançado em 2020, sendo uma extensão da linguagem GP também desenvolvida por Maloney (MALONEY; MÖNIG ;ROMAGOSA, 2021).

O que diferencia o MicroBlocks de outras ferramentas de programação em blocos é a combinação de programação com *feedback* imediato e uma execução de forma autônoma. Outras linguagens em blocos suportam um desses recursos, mas não ambos (MALONEY; MÖNIG ;ROMAGOSA, 2021).

Esta linguagem pode ser separada entre seu editor de blocos e sua máquina virtual. O editor permite ao usuário criar, editar e gerenciar códigos baseado em blocos e suas bibliotecas, é organizado de forma simples, separando os blocos em categorias e cores, assim como o Scratch. Inclui bibliotecas de sensores, redes, displays, servomotores, áudio e de seus kits educacionais suportados.

Figura 9 – Interface do MicroBlocks



Fonte: Adaptado de Maloney, Mönig e Romagosa (2021).

A Figura 9 demonstra as seções da interface de desenvolvimento da plataforma, onde pode-se destacar as categorias de blocos presentes nesta linguagem, o Quadro 2 apresenta as principais funcionalidades de cada categoria.

**Quadro 2 – Categorias de blocos do MicroBlocks**

| <b>Categorias</b> | <b>Funcionalidades</b>                                                                                                                                                      |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Saída             | Possui blocos para realizar saídas de sinais, como ligar um LED ( <i>Light-Emitting Diode</i> ), imprimir um valor na tela ou desenhar um sinal no gráfico.                 |
| Entrada           | Possui blocos para receber sinais vindos da placa de desenvolvimento, como botões, tempo de funcionamento e <i>timers</i> .                                                 |
| Pinos             | Funções para ler e escrever sinais em pinos selecionados, além da possibilidade de controlar pinos analógicos.                                                              |
| Comunicação       | A categoria de controle contém blocos que se comunicam com os periféricos da placa, como o I <sup>2</sup> C, SPI e UART.                                                    |
| Controle          | Possui diversos blocos para controle de lógica da programação, como <i>if</i> , <i>for</i> , <i>while</i> , <i>return</i> , <i>wait</i> e entre outros.                     |
| Operadores        | A categoria operadores agrupa os blocos para operações matemáticas, como soma, subtração, multiplicação e divisão, além de blocos para operações booleanas e de comparação. |
| Variáveis         | Possui funções para criação de variáveis globais e locais, além de funções para modificá-las.                                                                               |
| Dados             | A categoria dados contém blocos para criação e modificação de listas e vetores. Também possui funções para edição de <i>strings</i> .                                       |
| Meus Blocos       | Serve para que o usuário crie funções próprias, utilizando os blocos já existentes no programa.                                                                             |
| Bibliotecas       | Agrupa todas as bibliotecas oficiais disponibilizadas pelo MicroBlocks.                                                                                                     |

Fonte: Elaboração própria (2022).

O código em MicroBlocks é executado por um interpretador próprio. Sua máquina virtual é projetada para microcontroladores de 32 bits com pelo menos 16 Kb de RAM e 64 Kb de memória *Flash*, o Quadro 3 apresenta as placas suportadas pela plataforma.

**Quadro 3 – Placas suportadas pelo MicroBlocks**

| <b>Placas suportadas</b> |                             |
|--------------------------|-----------------------------|
| BBC micro:bit            | Citilab's ED1 (ESP32)       |
| Calliope mini            | Arduino Due (Parcialmente)  |
| AdaFruit Express         | Arduino Zero (Parcialmente) |
| AdaFruit Bluefruit       | M5Atom Matrix               |
| AdaFruit Clue            | NodeMCU                     |

|       |                            |
|-------|----------------------------|
| ESP32 | Raspberry Pi Pico (RP2040) |
|-------|----------------------------|

Fonte: Adaptado de Maloney, Mönig e Romagosa (2021).

O código em MicroBlocks é executado diretamente no microcontrolador, devido ao seu interpretador. À medida que se programa, o código é interpretado, baixado e armazenado de forma incremental na memória flash, caso o microcontrolador seja desconectado do ambiente de programação, o código estará salvo e será possível executar o programa diretamente na placa, possibilitando que os projetos sejam utilizados em baterias e ao ar livre, garantindo portabilidade e autonomia.

Pode-se destacar estas características importantes presentes na plataforma:

- *Feedback* imediato: É possível executar um bloco individualmente com apenas um clique, receber valores de sensores em tempo real e plotar sinais na ferramenta gráfica.
- Leitura do microcontrolador: Ao conectar novamente uma placa com scripts de MicroBlocks armazenados, fará com que os blocos utilizados reapareçam da mesma maneira que foram programados.
- Paralelismo: MicroBlocks permite escrever scripts separados para cada tarefa e executá-los simultaneamente.

O paralelismo acontece devido ao agendamento de tarefas, a máquina virtual executa algumas instruções do código e alterna em pontos pré-definidos para outro bloco de instruções, causando a impressão de estarem sendo executados paralelamente. Estes pontos pré-definidos encontram-se geralmente no final de um loop ou em uma instrução de espera (MALONEY; MÖNIG ;ROMAGOSA, 2021).

No entanto, a linguagem MicroBlocks também apresenta limitações, como a impossibilidade de converter seus blocos em uma linguagem de programação textual, dificultando a adaptação e aprendizado de novas linguagens. A linguagem suporta apenas números inteiros positivos e negativos, números decimais não podem ser usados, além disto, interrupções e *timers* disponíveis no microcontrolador não possuem blocos para acioná-los.

### 2.2.5 BIPES

BIPES é um acrônimo de plataforma integrada baseada em blocos para sistemas embarcados, consiste em um *software* de código aberto baseado em Blockly, desenvolvido no Brasil em cooperação com as universidades UFSCar (Universidade Federal de São Carlos), USP (Universidade de São Paulo), IFRN (Instituto Federal do Rio grande do Norte) e UFRN (Universidade Federal do Rio Grande do Norte). Seu desenvolvimento iniciou em 2020 e atualmente está na fase beta (JUNIOR et al., 2020).

O BIPES permite que qualquer pessoa projete, programe, construa, implante e teste de forma rápida e confiável sistemas embarcados e dispositivos IoT (JUNIOR et al., 2020). Faz uso das vantagens da programação em blocos aplicado na programação de sistemas embarcados, com o objetivo de facilitar o desenvolvimento de protótipos e auxiliar no aprendizado de novos desenvolvedores.

A plataforma executa diretamente no navegador em ambiente *web* sem *downloads* adicionais. É capaz de trabalhar com várias placas e microcontroladores, apresentadas no Quadro 4, usando linguagens de programação como Python, CircuitPython e MicroPython. O único requisito para o BIPES funcionar é o *firmware* do MicroPython ou CircuitPython estar previamente carregado no dispositivo que irá ser utilizado (JUNIOR et al., 2020).

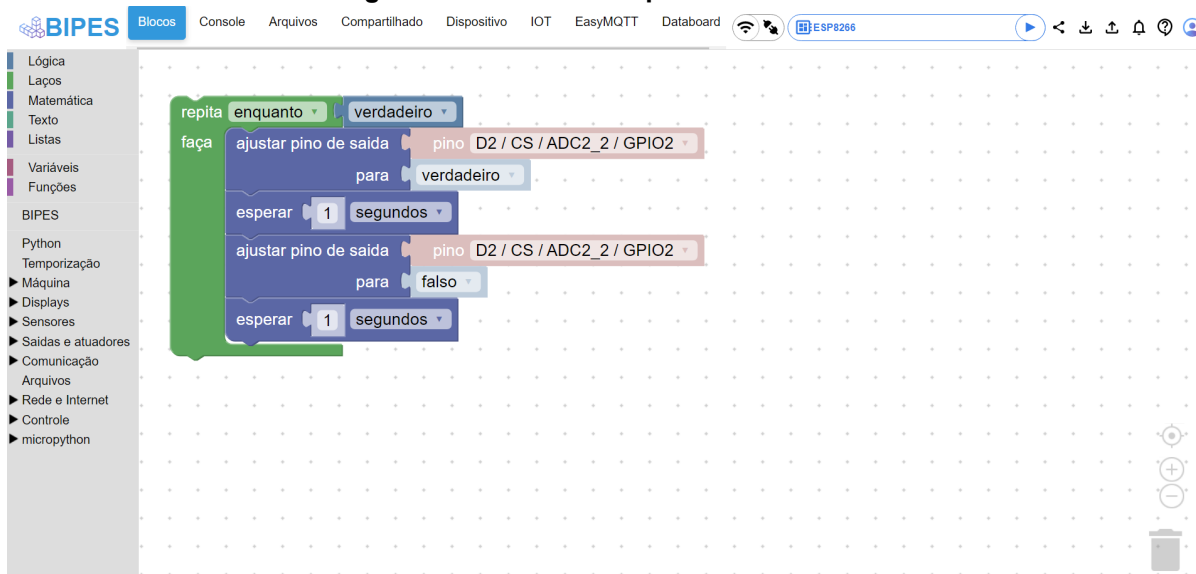
**Quadro 4 – Placas suportadas pelo BIPES**

| Placas suportadas |                                 |
|-------------------|---------------------------------|
| WEMOS D1 Mini     | stm32 / mbed                    |
| NodeMCU           | Arduino (UNO, Nano, Pro mini)   |
| ESP32             | MicroPython pyboard             |
| Raspberry Pi Pico | BeagleBone                      |
| bbc:microbit      | Raspberry Pi (1, 2, 3, 4, Zero) |

Fonte: Adaptado de BIPES (2022).

A plataforma possui um editor de blocos, semelhante ao Scratch, permite ao usuário criar funções, variáveis, editar e gerenciar códigos existentes e navegar pelas bibliotecas já implementadas, é organizado de forma simples como mostra a Figura 10, separando os blocos em categorias e cores.

Figura 10 – Interface da plataforma BIPES



Fonte: Elaboração própria (2022).

As categorias de blocos e suas funcionalidades são apresentados pelo Quadro 5.

Quadro 5 – Categorias de blocos do BIPES

| Categorias   | Funcionalidades                                                                                              |
|--------------|--------------------------------------------------------------------------------------------------------------|
| Lógica       | Possui blocos de lógica da programação, como <i>if</i> , comparações, verdadeiro e falso.                    |
| Laços        | Possui blocos para controle de laços, como <i>for</i> , <i>for each</i> , <i>while</i> e <i>break</i> .      |
| Matemática   | Agrupa os blocos para operações matemáticas, como soma, subtração, multiplicação, divisão e entre outros.    |
| Texto        | Funções para criação e modificação de <i>strings</i> .                                                       |
| Listas       | Funções para criação e modificação de listas.                                                                |
| Variáveis    | Possui funções para criação de variáveis globais, além de funções para modificá-las.                         |
| Funções      | Serve para que o usuário crie funções próprias, utilizando os blocos já existentes no programa.              |
| BIPES        | Blocos para armazenamento de dados sobre o projeto, como versionamento e assinatura.                         |
| Python       | Blocos para execução de códigos textuais escritos em Python.                                                 |
| Temporização | Possui funções para controle de tempo, como <i>delays</i> , contadores e <i>timers</i> .                     |
| Máquina      | Controle da leitura e escrita dos pinos, leitura e escrita de pinos analógicos e digitais e controle de PWM. |
| Displays     | Biblioteca interna para controle de displays amplamente utilizados.                                          |
| Sensores     | Biblioteca interna para controle de sensores amplamente utilizados.                                          |

|                    |                                                                             |
|--------------------|-----------------------------------------------------------------------------|
| Saídas e atuadores | Biblioteca interna para controle de relés, motores de passo e servomotores. |
| Comunicação        | Contém blocos que se comunicam com os periféricos da placa.                 |
| Arquivos           | Funções para leitura e escrita em arquivos                                  |
| Rede e internet    | Possui blocos para controle de comunicação com redes.                       |
| Controle           | Blocos para controle de PID ( <i>Proportional–Integral–Derivative</i> ).    |
| MicroPython        | Contém blocos para bibliotecas internas da linguagem MicroPython.           |

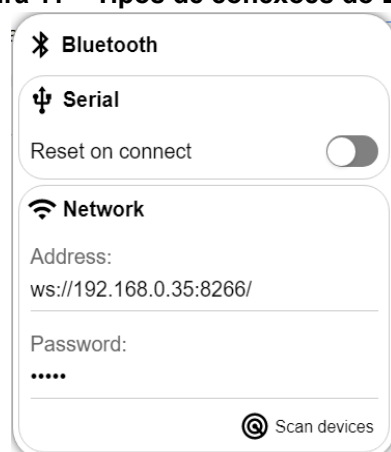
Fonte: Elaboração própria (2022).

Uma vez que o MicroPython e o CircuitPython são usados para projetos de IoT, o BIPES aproveita estes recursos fornecendo módulos para protocolos de comunicação, arquivos, sensores, atuadores, displays e redes. Além disso, existem blocos para operações com recursos internos do microcontrolador como interrupções, WDT (*Watchdog Timer*), *deep-sleep* e memória (JUNIOR et al., 2020).

A interface BIPES conta com uma aba para utilização do console, possibilitando a entrada e saída de dados, assim como uma linguagem textual padrão. Dispõe também uma aba de gerenciamento dos arquivos, permitindo aos usuários enviar arquivos locais diretamente para o dispositivo, que serão armazenados permanentemente no sistema do dispositivo. Esses arquivos também podem ser baixados da placa ou abertos no editor de texto nesta mesma guia (JUNIOR et al., 2020).

É possível fazer a conexão das placas de três formas distintas, sendo apresentadas pela Figura 11, está disponível a conexão serial através de USB (*Universal Serial Bus*), conexão WiFi (*Wireless Fidelity*) e *Bluetooth*.

**Figura 11 – Tipos de conexões do BIPES**

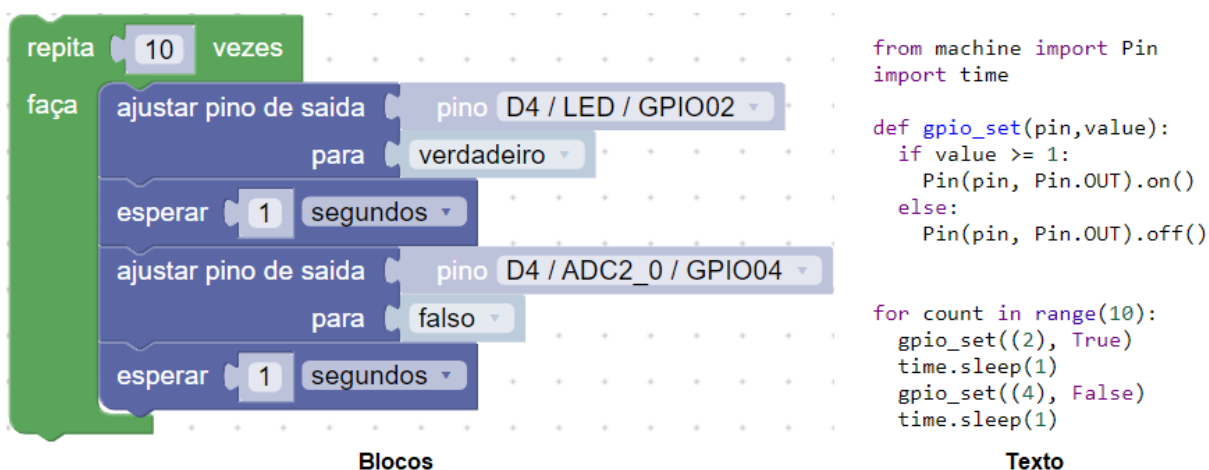


Fonte: BIPES (2022).

Para utilizar uma conexão sem fio, o dispositivo precisa ser configurado usando as ferramentas fornecidas pela plataforma, assim é possível programar em qualquer dispositivo móvel, como celular, tablet ou notebook e enviar o código programado nestes dispositivos diretamente para a placa, sem a necessidade de conectar um cabo (BIPES, 2022).

O código escrito em BIPES é traduzido automaticamente para MicroPython ou CircuitPython, como demonstrado pela Figura 12, assim que os blocos são adicionados na área de programação sua tradução em formato de linguagem textual fica disponível na aba de arquivos. Deste modo a linguagem se torna um híbrido de programação em blocos e textual, facilitando a adaptação e o aprendizado (BIPES, 2022).

**Figura 12 – Tradução do código em BIPES para MicroPython**

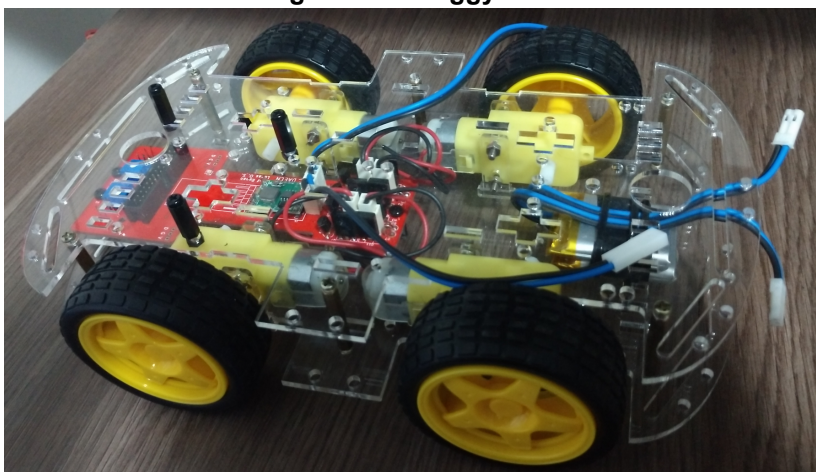


Fonte: Elaboração própria (2022).

## 2.3 Buggy AWD

O Buggy AWD apresentado na Figura 13, é um protótipo de veículo modular autônomo, originalmente construído por Lucas Odair Silva em 2019 como tema do Projeto Integrador 3, atualmente modificado pelo Prof. Renan Augusto Starke para ser usado como kit de desenvolvimento e ensino, está disponível no Laboratório de Pesquisa Avançada em Eletrônica do IFSC-Florianópolis (STARKE; LPAE, 2022).

**Figura 13 – Buggy AWD**



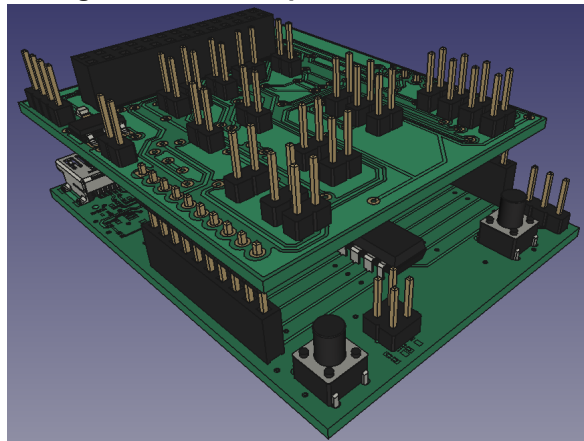
Fonte: Starke e LPAE (2022).

Este veículo é o dispositivo central deste trabalho, sendo responsável por fornecer um ambiente de estudo completo com módulos e sensores, para a utilização das ferramentas de programação em blocos. O *hardware* deste projeto pode ser dividido em três partes, o microcontrolador como núcleo do sistema, os módulos de sensores e o controle dos motores.

### 2.3.1 Microcontrolador

O cérebro consiste em um microcontrolador e seu conector de 26 vias que conecta os módulos e motores do veículo. Por se tratar de um projeto de ensino, foram desenvolvidas placas para fazer a comunicação dos microcontroladores com o Buggy, assim como a programação para fazer o controle. Até o momento foram elaboradas placas para o Arduino Uno, MSP-EXP430G2, apresentada pela Figura 27, MSP-EXP430FR2355 e Raspberry Pi. Este presente trabalho irá adicionar uma placa para o ESP32 Dev Kit v1, assim como sua programação em MicroPython.

**Figura 14 – Shield para MSP-EXP430G2**



Fonte: Starke e LPAE (2022).

### 2.3.2 Sensores e módulos

O Buggy possui sensores de encoder para aferição de velocidade, monitoramento para as baterias de LiPo (*Lithium Polymer Battery*), sensores infravermelho seguidores de linha, sensor de distância ultrassônico e módulo *Bluetooth* para comunicação com celulares e aplicativos de programação (STARKE; LPAE, 2022).

#### 2.3.2.1 Encoder

Para medir a velocidade é utilizado um encoder com sensor óptico, composto por um diodo emissor de luz e um receptor, separados por um disco com perfurações. A velocidade é calculada com base na variação do sensor óptico e devido aos 20 furos presentes no disco, como mostra a Equação 2.1.

$$Vm = n \cdot \frac{D \cdot \pi}{t \cdot 20} \quad (2.1)$$

Onde:

$V_m$  : velocidade em metros por segundo

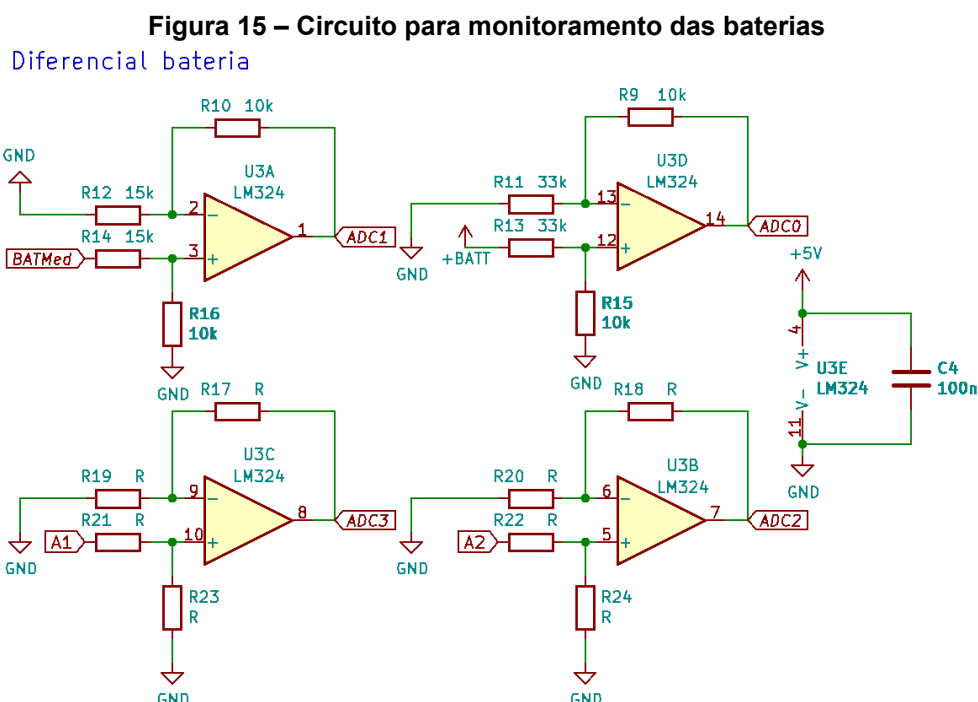
$n$  : número de pulsos por período

$D$  : diâmetro do disco em metros

$t$  : tempo entre medições em segundos

### 2.3.2.2 Monitoramento de baterias

Para garantir a segurança de seu funcionamento, duas baterias LiPo de 3,7 V são ligadas em série para gerar 7,4 V como fonte de energia principal para o Buggy. Para evitar a descarga abaixo de uma tensão segura, definiu-se 3 V como o limite para ambas as baterias. Para monitorar a tensão, o circuito diferencial com amplificador apresentado na Figura 15 é usado, com ganho de 1/3 para a tensão da bateria 1 e 2/3 para a tensão total em série. Isso permite a medição da tensão usando microcontroladores de baixa tensão.



Fonte: Starke e LPAE (2022).

### 2.3.2.3 Sensores seguidores de linha

O sensor seguidor de linha TCRT5000, apresentado na Figura 16, funciona com sensores de infravermelho, onde um LED emissor e um fototransistor de infravermelho são colocados lado a lado, definindo uma saída de baixo nível quando um obstáculo ou objeto passa dentro do alcance de ambos os sensores.

**Figura 16 – Sensor de linha TCRT5000**



Fonte: Vishay (2017).

#### 2.3.2.4 Sensor de distância ultrassônico

Para calcular a distância entre o Buggy e algum obstáculo é utilizado um sensor ultrassônico, que utiliza um emissor e um receptor de ultrassom, de modo que o tempo entre os sinais transmitidos e retornados pode ser usado para calcular a distância, como mostrada na Equação 2.2.

$$D = \frac{Pt \cdot c}{2} \quad (2.2)$$

Onde:

$D$  : distância

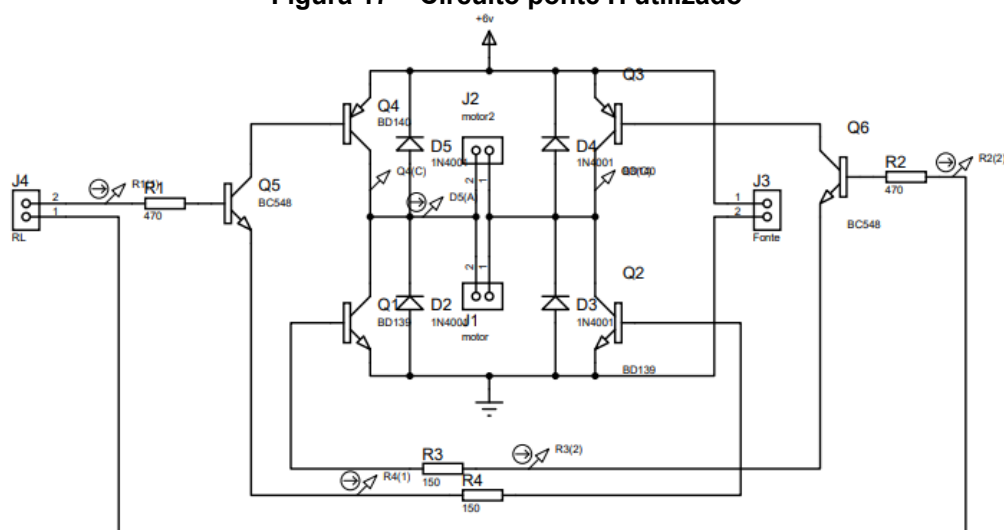
$Pt$  : tempo do pulso ultrassônico

$c$  : velocidade do som

#### 2.3.3 Motores

Para movimentar os motores, um circuito de ponte H é utilizado, pois permite mudar a direção do fluxo de corrente invertendo a tensão nos terminais dos transistores, o que permite o Buggy se mover em todas as direções. Como os motores são ligados em paralelo, é necessário o uso de duas pontes H para controlar os quatro motores.

**Figura 17 – Circuito ponte H utilizado**



Fonte: Starke e LPAE (2022).

A Figura 17 demonstra o circuito utilizado para fazer o controle de dois motores de uma única vez, com queda de tensão de carga de 0,4 V em relação a tensão de entrada e corrente de polarização externa de 640 pA, não exigindo uma corrente excessiva. É possível regular a velocidade do Buggy utilizando o PWM presente nos microcontroladores.

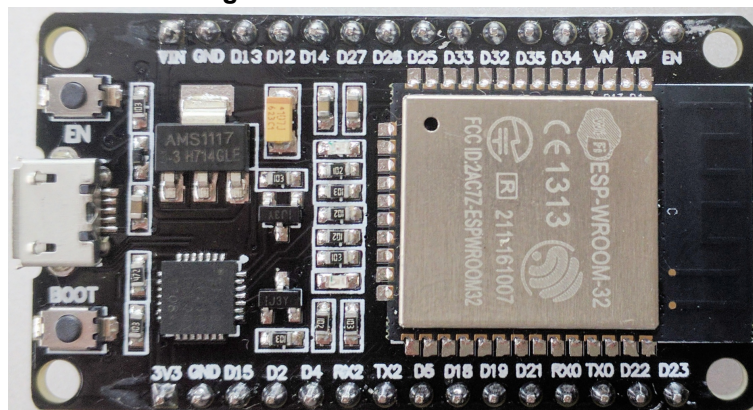
## 2.4 ESP32

Os microcontroladores ESP32 tem como grande característica fornecer *hardware* para o desenvolvimento de aplicações de IoT. Desta forma, possuem interface para WiFi 2,4 GHz e *Bluetooth*, além dos demais periféricos, como os protocolos de comunicação SPI, UART, I<sup>2</sup>C e CAN (*Controller Area Network*). Além disso, suportam uma ampla variedade de periféricos de entrada e saída, como toque capacitivo, ADC, DAC e PWM. (ESPRESSIF SYSTEMS, 2022).

Para a elaboração deste projeto, escolheu-se o kit de desenvolvimento ESP32 Dev Kit v1 para realizar o controle e a programação do Buggy. Alguns fatores foram decisivos para a escolha deste kit. O primeiro deles é a sua compatibilidade com a linguagem MicroPython e com ambas as ferramentas de programação em blocos MicroBlocks e BIPES, desta forma, facilitando a configuração destes ambientes.

Outro fator é a sua interface com WiFi e Bluetooth, possibilitando a utilização da programação sem fio da ferramenta BIPES. Por fim, a sua disponibilidade e custo benefício também tiveram uma grande participação na sua escolha.

**Figura 18 – ESP 32 Dev Kit v1**



Fonte: Espressif (2022).

Este kit, como apresenta a Figura 18, contém o microcontrolador ESP32-WROOM-32, que possui dois núcleos que podem ser controlados individualmente em uma frequência de *clock* de 80 MHz a 240 MHz. Contém 4 MB de memória *flash* e 520 KB de memória RAM, é operado por uma tensão de entrada de 3,3 V e tipicamente programado utilizando um cabo USB (ESPRESSIF SYSTEMS, 2020). Disponibiliza uma ampla quantidade de pinos e periféricos para uso, assim como detalhado pela Tabela 1.

**Tabela 1 – Pinos e periféricos do ESP32-WROOM-32**

| Pinos/Periféricos      | Quantidade |
|------------------------|------------|
| I/O de uso geral       | 34         |
| ADC 12-bits            | 18         |
| DAC 8-bits             | 2          |
| PWM 16-bits            | 16         |
| Sensor capacitivo      | 10         |
| UART 5Mbps             | 3          |
| SPI 40Mhz              | 4          |
| I <sup>2</sup> C 40Mhz | 2          |

Fonte: Elaboração própria (2022).

## 2.5 Resumo dos dispositivos e tecnologias adotados

Este capítulo apresentou os pontos mais importantes da pesquisa e do desenvolvimento deste trabalho. Foram demonstrados os recursos de linguagens de programação textuais e em blocos, bem como exemplos das linguagens atualmente usadas para a elaboração de projetos reais e protótipos para sistemas embarcados.

Destacou-se as diferenças das ferramentas de programação em blocos, sendo apresentadas pela Tabela 2.

**Tabela 2 – Características do MicroBlocks e BIPES**

| <b>Características</b>                           | <b>MicroBlocks</b> | <b>BIPES</b> |
|--------------------------------------------------|--------------------|--------------|
| Ferramenta em português                          | X                  | X            |
| Criação de bibliotecas/funções                   | X                  | X            |
| <i>Feedback</i> imediato                         | X                  |              |
| Paralelismo                                      | X                  |              |
| Interrupções                                     |                    | X            |
| Acesso direto ao microcontrolador                |                    | X            |
| Bibliotecas de rede, <i>bluetooth</i> e internet | X                  | X            |
| Bibliotecas de comunicação e I/O                 | X                  | X            |
| Números decimais                                 |                    | X            |
| Programação sem fio                              |                    | X            |
| Console                                          |                    | X            |
| Gráfico de dados                                 | X                  |              |
| Tradução de bloco para texto                     |                    | X            |
| Gerenciador de arquivos                          |                    | X            |

Fonte: Elaboração própria (2022).

Além disso, apresentou-se os principais recursos presentes no kit de desenvolvimento Buggy AWD, que será a base para a elaboração das bibliotecas utilizando as ferramentas de programação MicroBlocks e BIPES. O Quadro 6 apresenta os módulos empregados na elaboração das bibliotecas.

**Quadro 6 – Módulos utilizados nas bibliotecas**

| <b>Módulos</b>            |
|---------------------------|
| Monitoramento de baterias |
| Encoders de velocidade    |
| Controle dos motores      |
| Sensores de linha         |

|                                  |
|----------------------------------|
| Sensor de distância ultrassônico |
|----------------------------------|

Fonte: Elaboração própria (2022).

Por fim, é apresentado pela Tabela 3 os componentes e tecnologias que foram utilizados neste projeto e suas respectivas funções.

**Tabela 3 – Relação dos componentes e dispositivos adotados e sua respectiva função**

| <b>Componente ou tecnologia</b> | <b>Função</b>                                                                                                                                     |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| Buggy AWD                       | Dispositivo central deste trabalho, pois ele será responsável por fornecer o ambiente para a utilização das ferramentas de programação em blocos. |
| ESP32 Dev Kit v1                | Microcontrolador que será utilizado para realizar a programação e o controle do Buggy AWD.                                                        |
| MicroPython                     | Linguagem textual que será usada como base e referência para as bibliotecas de programação em blocos.                                             |
| MicroBlocks e BIPES             | Principais ferramentas para a elaboração das bibliotecas. Serão utilizadas para a programação em blocos do Buggy AWD.                             |

Fonte: Elaboração própria (2022).

### 3 METODOLOGIA

Esse trabalho tem como objetivo o estudo de ferramentas para o desenvolvimento de aplicações embarcadas, utilizando programação em blocos em conjunto com a implementação de bibliotecas de controle e o desenvolvimento do *hardware* de interface para o kit de robótica educacional Buggy AWD. Com isso, esta pesquisa pode ser classificada como aplicada, descritiva com abordagem exploratória e bibliográfica.

A pesquisa aplicada caracteriza-se por ter um objetivo prático, ou seja, seus resultados serão aplicados ou utilizados para atender a um interesse específico (MARCONI; LAKATOS, 2017). Baseando-se nisso, este estudo utilizou a pesquisa aplicada para desenvolver uma aplicação para o kit de ensino Buggy AWD utilizando ferramentas de programação em blocos.

Segundo Gil (2002, p. 41), uma pesquisa exploratória "tem como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses". Com relação à pesquisa descritiva, Gil (2002) descreve que este modelo de pesquisa é baseado nas correlações entre características ou variáveis que descrevem um fenômeno. Neste trabalho, são identificados os tipos de linguagens de programação textuais e em blocos, para que esse processo possa ser relevante para esse tipo de pesquisa.

Por fim, realizou-se uma pesquisa bibliográfica, ou seja, pesquisas utilizando artigos científicos e livros publicados (GIL, 2002). Neste estudo, artigos científicos, livros, monografias e materiais em meio eletrônico foram utilizados para construir a base de dados e os fundamentos teóricos para atingir os objetivos requeridos.

Neste trabalho, foi utilizado o *software* KiCad para o projeto de *layout* da placa de circuito impresso, utilizando como base as dimensões do kit de desenvolvimento ESP32 Dev Kit v1. Elaborou-se também, uma biblioteca aplicando a linguagem de programação MicroPython, para definir as funções mínimas que serão implementadas pelas linguagens de programação em blocos.

As ferramentas MicroBlocks e BIPES foram utilizadas para implementar as bibliotecas com programação em blocos. A partir da biblioteca base, desenvolvida

neste projeto, escrita em MicroPython, foram analisadas as características e limitações destas linguagens para o desenvolvimento da aplicação embarcada, identificando os elementos que precisam ser modificados ou removidos. Posteriormente, foram implementadas alterações necessárias em ambas as aplicações, descrevendo respectivamente o princípio de funcionamento de cada aplicação.

## 4 PROJETO E RESULTADOS

Este Capítulo é responsável por abordar o projeto dos componentes deste trabalho e os resultados obtidos. Primeiro, será descrito o projeto da placa de interface para a comunicação do microcontrolador com o Buggy AWD. Após isso, será apresentado o desenvolvimento das bibliotecas utilizando as tecnologias apresentadas no Capítulo 2. Por fim, será demonstrado a integração de todas as partes do projeto desenvolvido e os resultados obtidos.

### 4.1 Placa de interface

Para efetuar o controle do Buggy, projetou-se uma placa de interface para o microcontrolador ESP32 Dev Kit v1, que receberá as instruções das bibliotecas desenvolvidas neste projeto, comunicando-se com a placa de potência e com os módulos apresentados no Capítulo 2.

#### 4.1.1 Placa de potência e de módulos do Buggy

Com o intuito de projetar a placa de interface para o ESP 32, deve-se considerar a placa de potência do Buggy AWD. Esta placa faz a união de todos os módulos e sensores presentes no veículo, disponibilizando um barramento de 26 pinos para a conexão com as placas de interface projetadas.

Devido a conexão de controle, os sinais podem ser recebidos e enviados diretamente para o microcontrolador, sem a necessidade de componentes adicionais na placa de interface, apenas circuitos de proteção para os pinos do ESP 32.

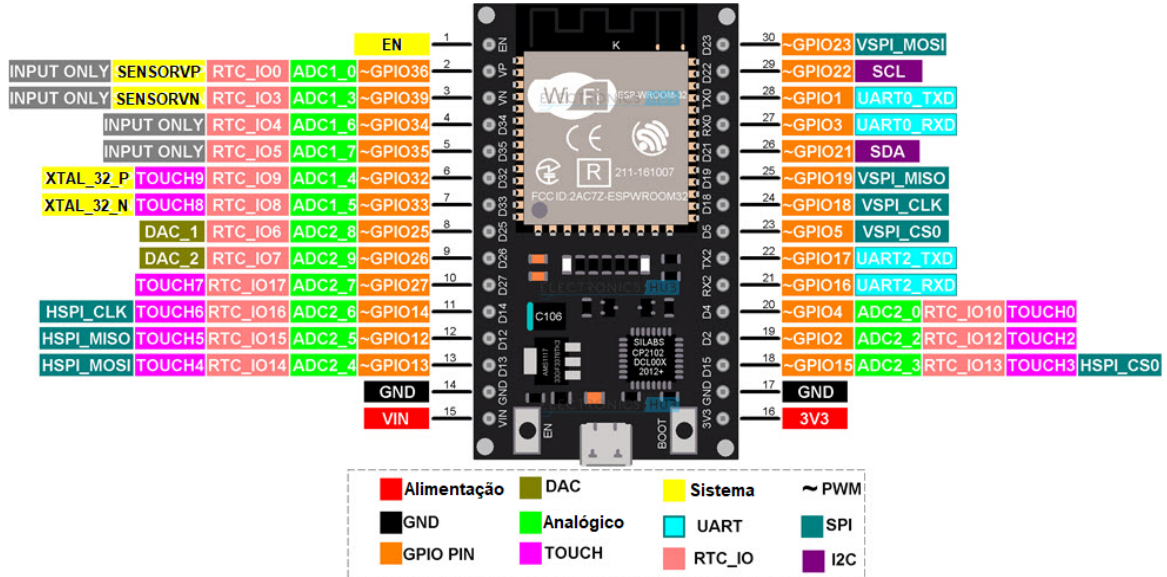
Como apresentado pela Figura 19, a conexão de controle disponibiliza para o microcontrolador uma tensão de 5 V regulada a partir das duas baterias ligadas em série, utilizando um regulador de tensão do tipo *Step-up-Down* que permite correntes de até 5 A. Esta tensão também é utilizada nos módulos e sensores presentes na placa.

A conexão de controle possibilita o acesso aos *encoders* ópticos de velocidade, sensores de distância HCSR04 ultrassônicos, sensores de linha



placa de potência, aplicando divisores de tensão e diodos BAT54S para garantir uma tensão máxima de 3,3 V nos seus terminais.

**Figura 20 – Pinos do ESP 32 Dev Kit v1**



Fonte: Adaptado de Teja (2022).

As conexões realizadas entre o ESP 32 Dev Kit v1 e a placa de potência são demonstradas na Tabela 4.

**Tabela 4 – Conexões com entre o ESP 32 e a placa de potência**

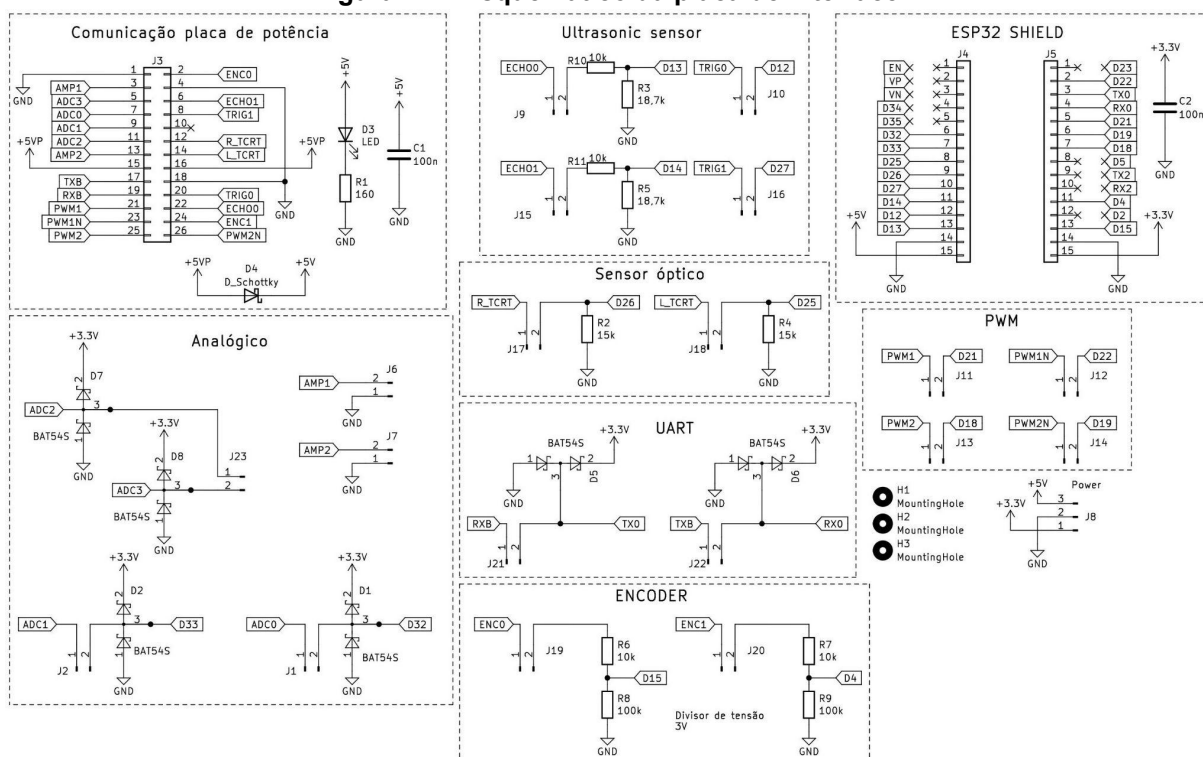
| Pinos     | Sinal                                   |
|-----------|-----------------------------------------|
| VIN       | Tensão 5 V da placa de potência         |
| GND       | Referência da placa de potência         |
| D32 (ADC) | Bateria 1 (Tensão da bateria 1)         |
| D33 (ADC) | Bateria 2 (Tensão total das baterias)   |
| D13       | Echo dianteiro (Sensor de distância)    |
| D12       | Trigger dianteiro (Sensor de distância) |
| D14       | Echo traseiro (Sensor de distância)     |
| D27       | Trigger traseiro (Sensor de distância)  |
| D21 (PWM) | Motor 1                                 |
| D22 (PWM) | Motor 1 negativo                        |
| D18 (PWM) | Motor 2                                 |
| D19 (PWM) | Motor 2 negativo                        |
| D15       | Encoder 0 (Sensor de velocidade)        |
| D4        | Encoder 1 (Sensor de velocidade)        |
| D26       | Sensor ótico direita (Sensor de linha)  |
| D25       | Sensor ótico esquerda (Sensor de linha) |
| TX (UART) | Transmissor <i>Bluetooth</i>            |
| RX (UART) | Receptor <i>Bluetooth</i>               |

Fonte: Elaboração própria (2022).

### 4.1.3 Esquemático, layout e modelo 3D

A placa de interface foi projetada utilizando o *software* KiCad. Adicionou-se *jumpers* entre os sinais de acionamento e dos sensores, como apresenta a Figura 21 para facilitar a desconexão e possível verificação dos sinais durante a elaboração das aplicações.

Figura 21 – Esquemático da placa de interface



Fonte: Elaboração própria (2022).

Nesse circuito utilizou-se um diodo (D4) para proteção contra tensão reversa, originada pela conexão USB do kit ESP 32, também inclui-se um LED (D3) para indicar se a tensão proveniente da placa de potência está sendo recebida. Foram inseridos dois capacitores de filtro de alta frequência (C1 e C2) como boas práticas no desenvolvimento do esquemático.

Seguido do desenvolvido do esquemático, projetou-se o layout da PCB (*Printed Circuit Board*), presente na Figura 22. Durante esse processo, também foram seguidas as boas práticas de desenvolvimento de um layout de PCB, fazendo a utilização de planos de referência e evitando as vias entre os planos.



## 4.2 Construção dos códigos

A construção dos códigos começou pela biblioteca base da linguagem textual MicroPython, servindo como referência para as linguagens em blocos MicroBlocks e BIPES. Inclui-se todas as funções presentes no kit de desenvolvimento Buggy AWD, como as medidas de distância, aferição da velocidade, controle dos motores, das baterias e dos sensores de linha.

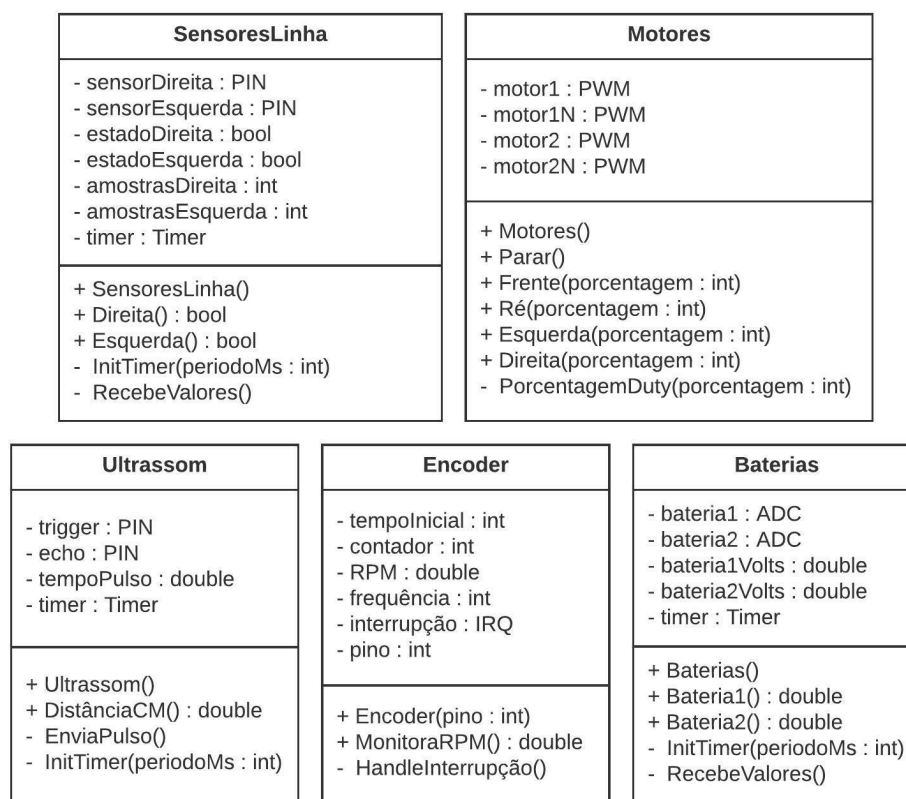
### 4.2.1 MicroPython

O código em MicroPython é importante para a construção deste projeto, será a base para o desenvolvimento de códigos em ambas as linguagens de programação em blocos. Ele estabelece as funções e soluções mínimas a serem implementadas por essas linguagens, servindo como um ponto de partida.

Para a concepção da programação em MicroPython utilizou-se o software de desenvolvimento Thonny IDE (*Integrated Development Environment*), pois disponibiliza ferramentas para gravação de *firmware*, sistema de gerenciamento de arquivos, *highlight* de código e um *debugger*, facilitando os testes e configurações da placa.

Utilizou-se a ferramenta de gravação de *firmware* do Thonny, através da porta USB do kit ESP 32 gravando o *firmware* do MicroPython na versão estável 1.19.1, mais atual até o momento deste trabalho. Para a gravação é necessário a instalação do driver USB para UART, disponível no site oficial do MicroPython.

O desenvolvimento do código baseou-se nos paradigmas de programação orientada a objetos, possibilitando o encapsulamento e a organização dos códigos, tornando-os mais legíveis e facilitando a sua reutilização entre os objetos. Os módulos e suas implementações foram divididos em classes como apresenta a Figura 24.

**Figura 24 – Diagrama UML de classes das classes de controle do Buggy AWD**

Fonte: Elaboração própria (2022).

A biblioteca de códigos em MicroPython está disponível na seção de apêndices. As principais responsabilidades exercidas por cada classe demonstrada na Figura 24 estão detalhadas pela Tabela 5.

**Tabela 5 – Relação das classes do controle e suas respectivas responsabilidades**

| Classe        | Responsabilidade                                                                                                                                                                                                               |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SensoresLinha | Sua responsabilidade é receber os sinais dos sensores de linha do Buggy AWD, tratando-os de forma que não haja um falso positivo, fornecendo métodos de retorno para os sensores de ambos os lados do veículo.                 |
| Motores       | Esta classe faz o controle dos motores utilizando PWM, disponibilizando métodos para a movimentação do Buggy em todas as direções.                                                                                             |
| Ultrassom     | Tem como objetivo controlar os sensores de distância HCSR04, enviando pulsos de som e aguardando pelo seu retorno, a fim de se estimar a distância aproximada dos obstáculos, fornecendo métodos para retorno desta distância. |
| Encoder       | Esta classe é utilizada para aferição da velocidade do Buggy, realizando o cálculo de RPM (Rotações Por Minuto) através do sensor ótico e disponibilizando um método para seu retorno.                                         |
| Baterias      | Tem como responsabilidade o monitoramento de tensão das baterias presentes no Buggy, utilizando o ADC, fornecendo as tensões em métodos de retorno.                                                                            |

Fonte: Elaboração própria (2022).

A classe SensoresLinha recebe o valor do módulo sensor de linha a cada 5 milissegundos através de um *timer* interno do ESP 32, que é iniciado no construtor da classe. É apresentado pela Figura 25 a estratégia de *debounce* por contagem, para garantir que não haja sinais indesejados, realizando uma contagem de 10 sinais, para confirmar seu estado.

**Figura 25 – Funcionamento da classe SensoresLinha**

```

1.  def _init_timer(self, period_ms = 5):
2.      self.tim1.init(period=period_ms, mode=Timer.PERIODIC,
3.      callback=self._recebe_valores)
4.  def _recebe_valores(self, t):
5.      if (self.sensor_direita.value() != self.estado_direita):
6.          # Se contagem = 0, debounce terminado
7.          self.amostras_direita -= 1
8.          if (self.amostras_direita <= 0):
9.              self.estado_direita = self.sensor_direita.value()
10.         else:
11.             self.amostras_direita = self.CONTAGEM
12.
13.         if (self.sensor_esquerda.value() != self.estado_esquerda):
14.             # Se contagem = 0, debounce terminado
15.             self.amostras_esquerda -= 1
16.             if (self.amostras_esquerda <= 0):
17.                 self.estado_esquerda = self.sensor_esquerda.value()
18.             else:
19.                 self.amostras_esquerda = self.CONTAGEM

```

Fonte: Elaboração própria (2022).

O funcionamento da classe Motores ocorre através do controle dos PWMs presentes no ESP 32, o usuário indica uma porcentagem de velocidade que é convertida para uma resolução de 10 bits como demonstra a Figura 26. Assim é alterado o *duty cycle* dos PWMs que por sua vez atuam na ponte H, acelerando ou parando o veículo.

**Figura 26 – Funcionamento da classe Motores**

```

1.  def _porcentagem_duty(self, porcentagem):
2.      return int((porcentagem * 1023) / 100)
3.
4.  def mover_frente(self, porcentagem):
5.      self.parar()
6.      if (porcentagem > 0) and (porcentagem <= 100):
7.          duty = self._porcentagem_duty(porcentagem)
8.          self.m1.duty(duty)
9.          self.m2.duty(duty)

```

Fonte: Elaboração própria (2022).

Para determinar a velocidade, a classe Encoder realiza por meio de interrupções a leitura dos sinais originados dos encoders, posicionados nas rodas do Buggy, os métodos desenvolvidos, ilustrado pela Figura 27, demonstram que cada interrupção causada pelas perfurações no disco do encoder geram um acréscimo no contador de revoluções. Utilizando a Equação 2.1 apresentada no Capítulo 2 é possível calcular a frequência no período de tempo determinado em que o contador sofreu o acréscimo das interrupções, tornando possível o cálculo das rotações por minuto.

**Figura 27 – Funcionamento da classe Encoder**

```

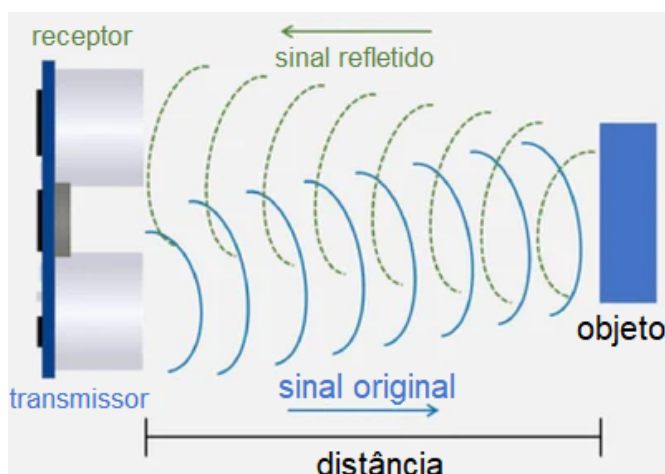
1.  def handle_interrupt(self, channel):
2.      self.counter +=1
3.
4.  def monitorRPM(self):
5.      if (time() - self.time0) > self.INTERVALO:
6.          state = machine.disable_irq()
7.          self.freq = self.counter/(time() - self.time0)
8.          self.rpm = self.freq/self.IRQ*60
9.          self.counter = 0 # Reseta o counter para a próxima revolução
10.         self.time0 = time() # Reseta o timer para próximo intervalo
11.         machine.enable_irq(state)
12.         return self.rpm

```

Fonte: Elaboração própria (2022).

O sensor ultrassônico HCSR04 é baseado no princípio de medição de tempo de reflexão de sinais sonoros, ele é composto por dois componentes principais, o transmissor e o receptor, apresentados na Figura 28. O transmissor envia um pulso sonoro e, quando esse sinal é refletido por um objeto, é captado pelo receptor. A partir da medição do tempo de ida e volta do sinal, é possível calcular a distância entre o objeto e o sensor, levando em consideração a velocidade do som no ar.

**Figura 28 – Funcionamento do sensor ultrassônico**



Fonte: Adaptado de Robocraze (2022).

A classe *Ultrassom* utiliza um *timer* periódico de 500 milissegundos, da mesma forma que a classe *SensoresLinha*. Observa-se pelos métodos ilustrados na Figura 29, que o sinal *trigger* é enviado durante 10 microsegundos, aguardando até receber o retorno deste sinal pelo sensor, armazena-se o tempo de espera na variável *pulse\_time*, que é utilizada no método público *distancia\_cm*, para retornar a distância aproximada em centímetros do obstáculo.

**Figura 29 – Funcionamento da classe *Ultrassom***

```

1.  def _init_timer(self, period_ms = 500):
2.      self.tim0.init(period=period_ms, mode=Timer.PERIODIC,
   callback=self._envia_pulso_e_espera)
3.
4.  def _envia_pulso_e_espera(self, t):
5.      self.trigger.value(0)
6.      time.sleep_us(5) # Aguarda a estabilização do sensor
7.      self.trigger.value(1) # Envia pulso por 10 us
8.      time.sleep_us(10)
9.      self.trigger.value(0)
10.     self.pulse_time = machine.time_pulse_us(self.echo, 1,
   self.ECHO_TIMEOUT)
11.
12.  def distancia_cm(self):
13.      # pulse_time / 2 (o pulso vai e volta) e dividido
14.      # por 29,13 porque a velocidade do som no ar é
15.      # 0.034320 cm/us que é 1 cm cada 29.1375us
16.      cms = (self.pulse_time / 2) / 29.1375
17.      return cms

```

Fonte: Elaboração própria (2022).

O monitoramento das baterias é feito pela classe Baterias que utiliza o ADC interno de 12 bits do ESP 32. O processo de leitura dos valores de 0 V até 3,3 V, apresentado nos métodos da Figura 30, são realizados devido à placa de potência possuir o circuito de ampops apresentado no Capítulo 2, sendo necessário multiplicar o valor da leitura por um fator de conversão, para obter a medida correta de tensão das baterias de 3,7 V.

**Figura 30 – Funcionamento da classe Baterias**

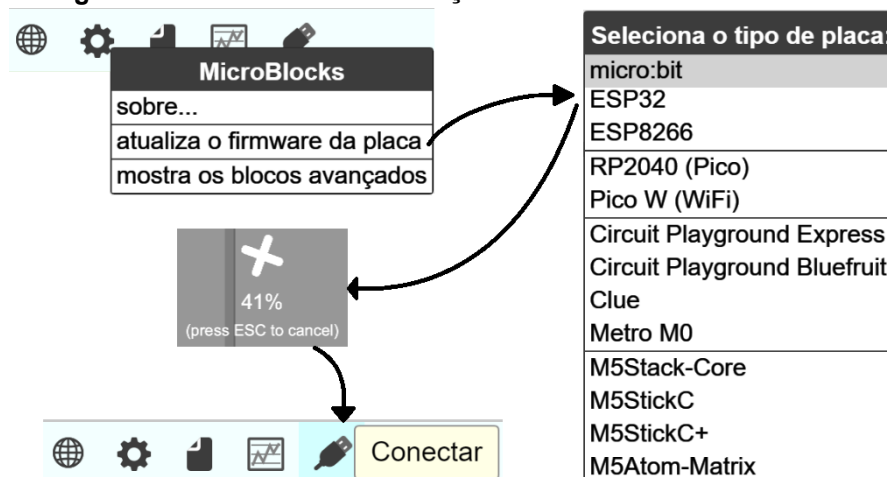
```
1. def _recebe_valores(self, t):
2.     self.bat1_volts = self.bat1.read()
3.     self.bat2_volts = self.bat2.read()
4.
5. def _init_timer(self, period_ms = 5000):
6.     self.tim2.init(period=period_ms, mode=Timer.PERIODIC,
7.         callback=self._recebe_valores)
8.
9. def bateria1(self):
10.    return self.bat1_volts * self.FATOR_ADC
```

Fonte: Elaboração própria (2022).

#### 4.2.2 MicroBlocks

Para realizar a programação do ESP 32 em MicroBlocks é necessário a instalação do seu *firmware*, disponível na sua interface de desenvolvimento. A Figura 31 demonstra o processo necessário para esta tarefa, com o microcontrolador conectado à porta USB do computador, abre-se as configurações da plataforma, seleciona-se a placa para a instalação, aguarda-se o processo terminar e ao fim a placa estará configurada e pronta para ser conectada ao ambiente de programação.

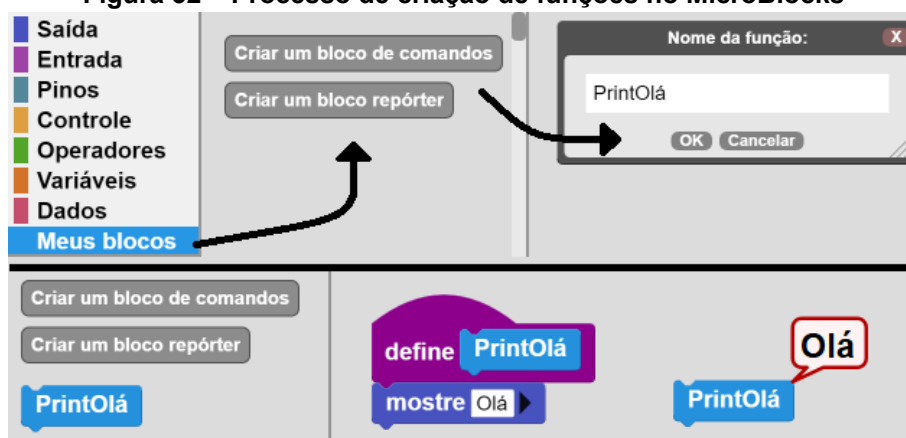
**Figura 31 – Processo de instalação do *firmware* do MicroBlocks**



Fonte: Elaboração própria (2022).

Após a configuração do microcontrolador, habilitou-se a opção de blocos avançados presente na Figura 31 que permite a criação e exportação de bibliotecas de código. É apresentado pela Figura 32 o processo de criação de funções na plataforma, sendo possível criar blocos de comandos e de repórter. Blocos de comandos servem para realizar um comando específico sem retorno, já os blocos repórter podem realizar comandos e serem utilizados como parâmetro em outros blocos, retornando algum valor.

**Figura 32 – Processo de criação de funções no MicroBlocks**



Fonte: Elaboração própria (2022).

O desenvolvimento em MicroBlocks é baseado em bibliotecas de funções, assim não é possível implementar o conceito de programação orientada a objetos e criação de classes como em MicroPython.

A biblioteca "Buggy" foi criada para controlar os módulos do veículo de forma organizada. Ela contém funções agrupadas e pode ser compartilhada com outros usuários. As interfaces de cada função desenvolvida são apresentadas pela Figura 33. A biblioteca "Buggy" foi criada especificamente para controlar o Buggy, partindo da base de código criada com MicroPython.

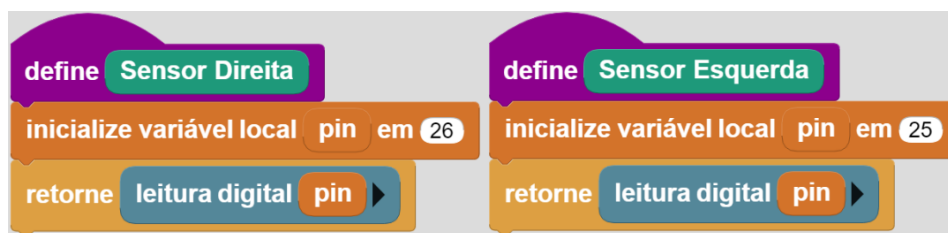
**Figura 33 – Interface das funções desenvolvidas no MicroBlocks**



Fonte: Elaboração própria (2022).

Para o controle dos sensores de linha, foram criadas funções do tipo repórter com nome Sensor Direta e Sensor Esquerda. Devido a incapacidade do MicroBlocks em lidar com os *timers* do ESP 32, retorna-se o valor do sensor de maneira direta, sem a utilização de um *debounce*, a definição das funções é mostrada na Figura 34.

**Figura 34 – Definição das funções dos sensores de linha no MicroBlocks**



Fonte: Elaboração própria (2022).

O controle dos motores é realizado por funções do tipo comandos, recebendo um valor inteiro de 0 a 100 que representa a porcentagem de *duty cycle* que será utilizada na ponte H. No entanto, valores de *duty cycle* abaixo de 50% e baterias com carga insuficiente não geram torque suficiente para movimentar o Buggy. Mesmo assim, o usuário pode ajustar a velocidade para qualquer valor desejado.

A Figura 35 demonstra a definição da função Mover Frente, que converte o valor percentual em uma resolução de 10 bits para fazer o controle do PWM presente nos pinos selecionados. A função Parar Motores funciona da mesma maneira parando os motores com uma razão cíclica de 0.

**Figura 35 – Definição das funções dos motores no MicroBlocks**



Fonte: Elaboração própria (2022).

Para realizar a aferição da velocidade do veículo, foi utilizado apenas um dos encoders presentes, utilizou-se o paralelismo disponível no MicroBlocks para fazer a contagem das revoluções, sendo que o mesmo não permite a implementação de interrupções, assim evitou-se o bloqueio de operações devido aos laços infinitos utilizados. A Figura 36 apresenta a definição e operação das funções relacionadas ao encoder.

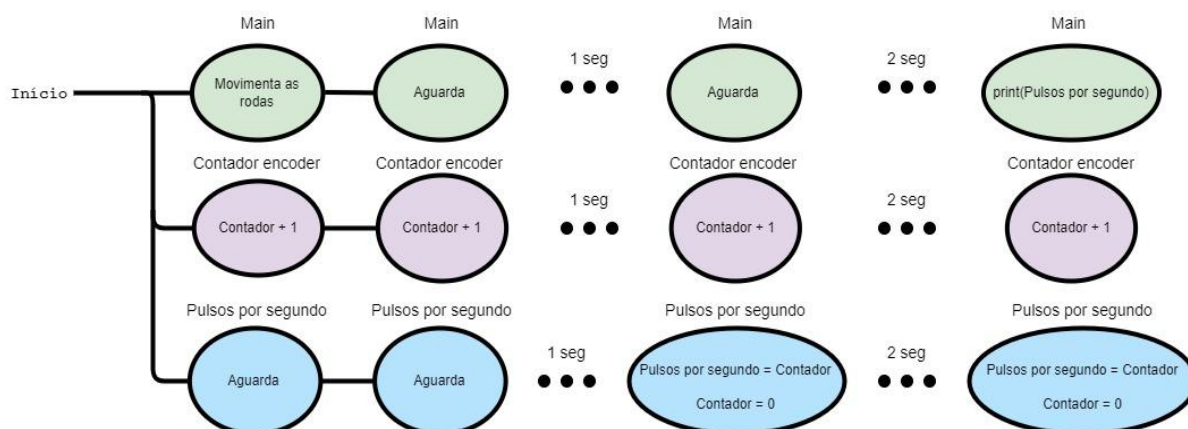
**Figura 36 – Exemplo de funcionamento e definição das funções do encoder no MicroBlocks**



Fonte: Elaboração própria (2022).

Ao iniciar o programa, a função principal faz com que a roda gire e o encoder do veículo comece a enviar pulsos de revolução. A função Contador Encoder, executada em paralelo, lê e armazena as contagens em uma variável global. A função Pulsos Por Segundo, também em execução paralela, aguarda um segundo, registra o valor da variável contador na variável global pulsos por segundo e zera a contagem. Dessa forma, o usuário pode acessar a variável pulsos por segundo para obter o número de pulsos recebidos pelo encoder em um segundo. A Figura 37 demonstra diagrama de tempo do exemplo citado.

**Figura 37 – Diagrama de tempo do funcionamento das funções do encoder no MicroBlocks**

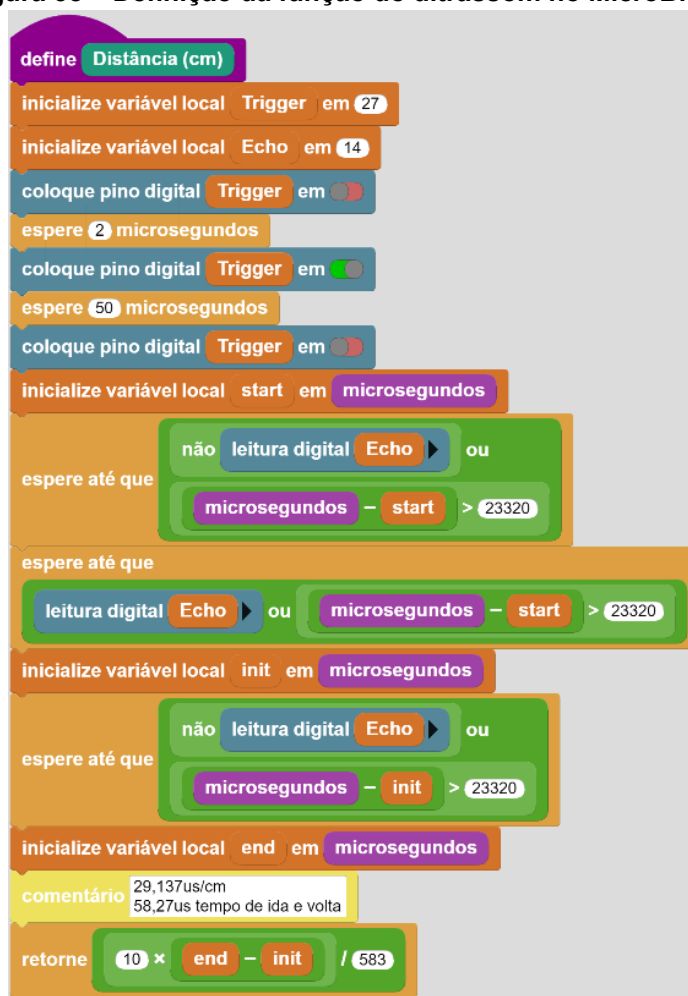


Fonte: Elaboração própria (2022).

O cálculo de RPM não é possível no MicroBlocks devido à ausência de suporte para números decimais. Ao realizar o cálculo de frequência como descrito no MicroPython, presente na Figura 27, o denominador da divisão sendo o intervalo de tempo em milissegundos é muito grande em relação ao numerador, que é a quantidade de interrupções. Como resultado, a divisão sempre gera um número decimal menor do que um, que é arredondado para zero.

A Figura 38 apresenta a definição da função repórter dos sensores de distância, que segue o mesmo padrão da Equação 2.2. Um sinal trigger é enviado durante 50 microsegundos, aguarda-se até receber o retorno deste sinal, caso o sinal seja recebido novamente até o tempo limite de 23320 microsegundos, limite de 4 metros do sensor, será finalmente calculado a distância em centímetros, deve-se considerar o fato do MicroBlocks não utilizar valores decimais para os cálculos, diminuindo a precisão dos cálculos.

**Figura 38 – Definição da função do ultrassom no MicroBlocks**



O controle de tensão das baterias é realizado pelas funções repórter retratadas na Figura 39. Devido a ausência de suporte para cálculos com valores decimais, não é possível realizar o cálculo do fator de correção, pelo mesmo motivo apresentado nos encoders, a divisão gera um valor menor que um, que é arredondado para zero. As funções retornam apenas o valor da leitura em resolução de 10 bits, sendo 1023 equivalente a 3,3 V e 0 igual a 0 V.

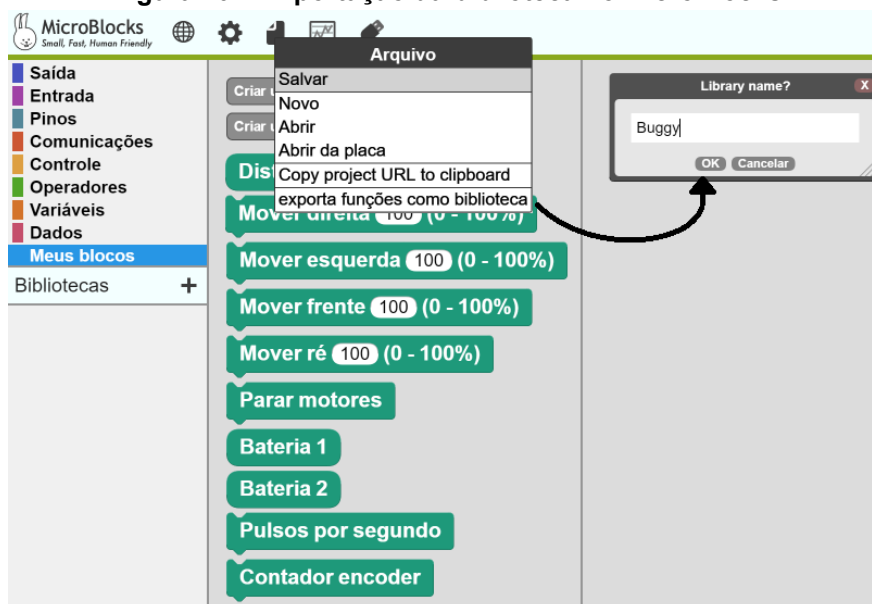
**Figura 39 – Definição das funções de bateria no MicroBlocks**



Fonte: Elaboração própria (2022).

Ao finalizar o desenvolvimento da programação em MicroBlocks, exportou-se todas as funções criadas utilizando a ferramenta de exportação da interface como ilustrado pela Figura 40. Assim é possível armazenar e compartilhar a biblioteca de funções, importando para o MicroBlocks pelo botão “mais” na categoria Bibliotecas.

**Figura 40 – Exportação da biblioteca no MicroBlocks**



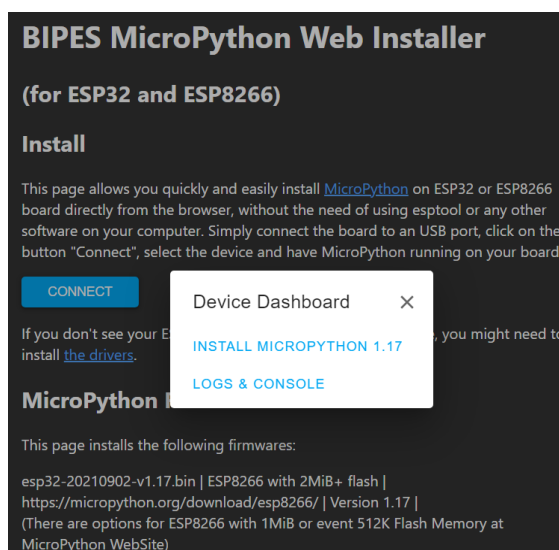
Fonte: Elaboração própria (2022).

### 4.2.3 BIPES

Para realizar a programação do ESP 32 no BIPES, é necessário a instalação do *firmware* do MicroPython. Utiliza-se o mesmo modo apresentado na Seção 4.2.1, empregando o software Thonny IDE para a instalação do *firmware*, ou aplicando de maneira *online* com a ferramenta web disponível na interface de desenvolvimento do BIPES, removendo a necessidade de baixar o *firmware* localmente e o software Thonny.

É demonstrado pela Figura 41 o processo necessário para a instalação na ferramenta *web*. Com o microcontrolador conectado à porta USB do computador, pressiona-se no botão “*Connect*”, seleciona-se a placa e aprova-se a instalação, aguarda-se o processo terminar, ao fim, a placa estará configurada e pronta para ser conectada ao ambiente de programação.

**Figura 41 – Plataforma de instalação *online* do MicroPython no BIPES**



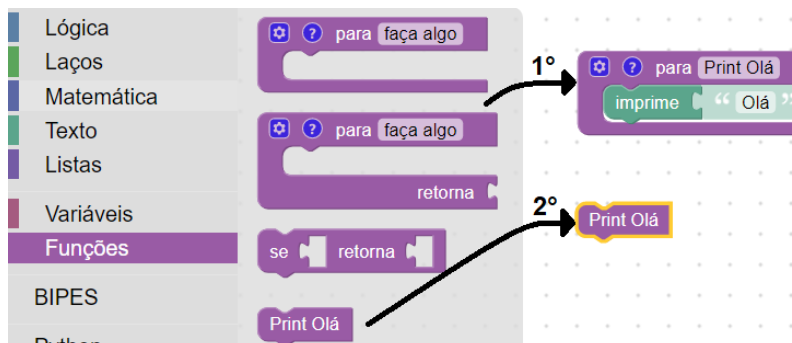
Fonte: Elaboração própria (2022).

Após a instalação do *firmware*, é possível conectar o microcontrolador na ferramenta de desenvolvimento BIPES utilizando as três maneiras apresentadas no Capítulo 2, sendo via USB, Bluetooth e WiFi. Com a placa conectada via USB, deu-se início a criação das funções para controle dos módulos e sensores.

A Figura 42 apresenta o processo de criação de funções na plataforma, sendo possível criar funções do tipo *void* e funções com retorno. As funções com

retorno são utilizadas somente como parâmetros para outras funções, mas podem realizar comandos em seu escopo.

**Figura 42 – Processo de criação de funções no BIPES**



Fonte: Elaboração própria (2022).

O desenvolvimento em BIPES é baseado em funções, impossibilitando a aplicação do paradigma de programação orientada a objetos e criação de classes, mesmo sendo baseado em MicroPython. Para a organização do controle dos módulos do Buggy, foram criadas funções organizadas em apenas um projeto, que pode ser compartilhado e utilizado por outros usuários. É ilustrado pela Figura 43 as interfaces de cada função desenvolvida.

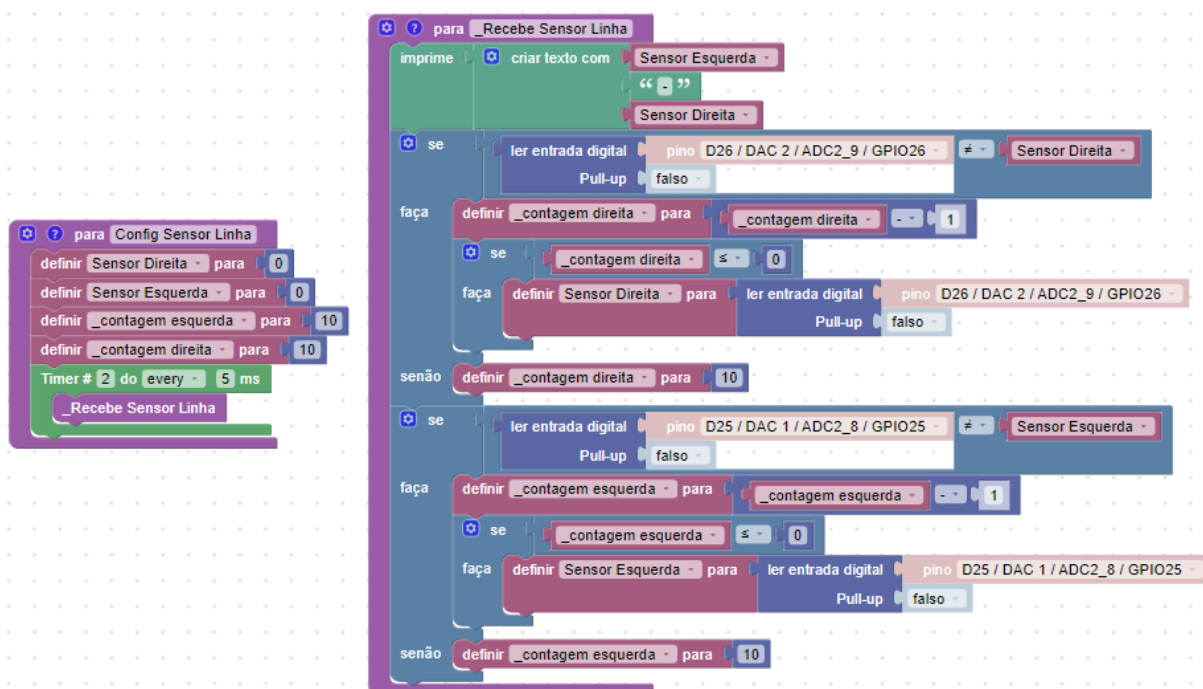
**Figura 43 – Interface das funções desenvolvidas no BIPES**



Fonte: Elaboração própria (2022).

O controle dos sensores de linha são feitos pelo *timer* interno, que recebe o sinal dos sensores a cada 5 milissegundos, necessitando ser iniciado pela sua função de configuração. Empregou-se a estratégia de *debounce* por contagem para garantir que não haja sinais indesejados, realizando uma contagem de 10 sinais, como mostra a Figura 44.

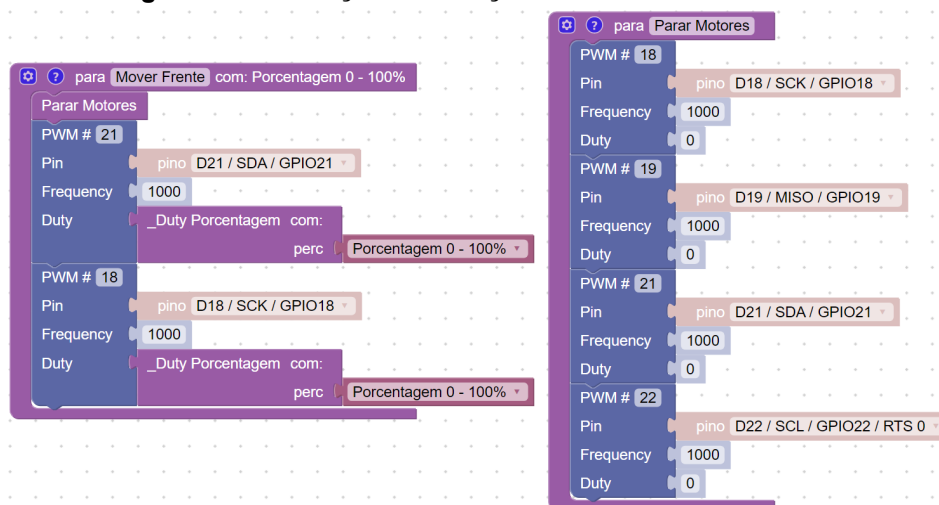
**Figura 44 – Definição das funções dos sensores de linha no BIPES**



Fonte: Elaboração própria (2022).

O funcionamento dos motores ocorre através da modulação dos PWM. Suas funções recebem como parâmetro uma porcentagem que é convertida em uma resolução de 10 bits pela função *Duty Porcentagem* como demonstra a Figura 45. Assim é alterado o *duty cycle* dos PWMs, que por sua vez atuam na ponte H, acelerando ou parando o Buggy.

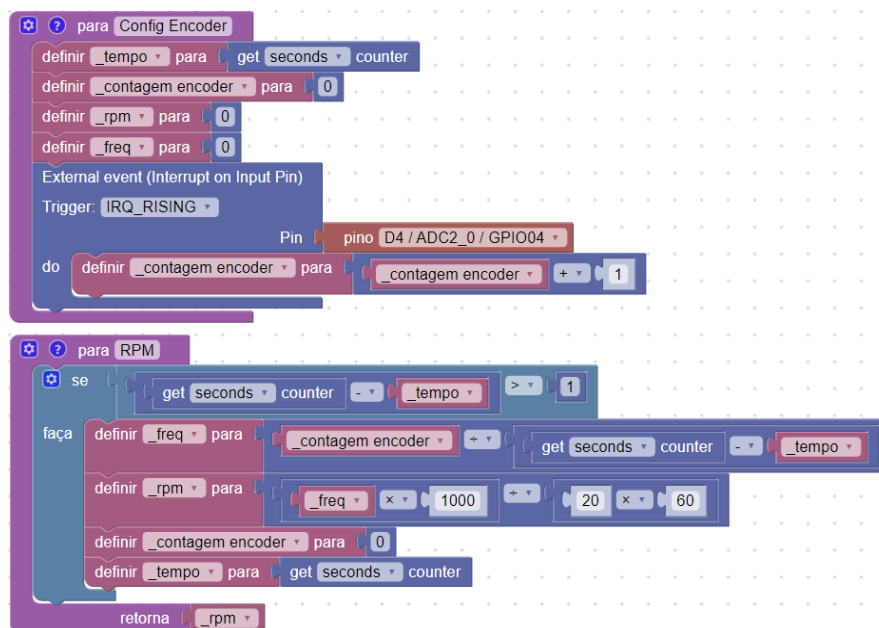
**Figura 45 – Definição das funções dos motores no BIPES**



Fonte: Elaboração própria (2022).

Para determinar a velocidade de uma roda, implementou-se interrupções em cada leitura dos sinais originados do encoder. As funções desenvolvidas são apresentadas pela Figura 46, onde cada interrupção gera um acréscimo na variável contadora de revoluções. Utilizando a mesma solução realizada na Classe Encoder do MicroPython, calcula-se as rotações por minuto no instante.

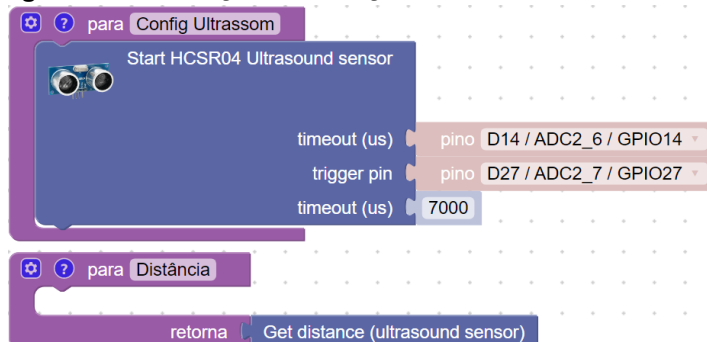
**Figura 46 – Definição das funções do encoder no BIPES**



Fonte: Elaboração própria (2022).

O controle dos sensores de distância é realizado através da biblioteca de sensores já existente na plataforma, sendo apenas desenvolvido uma interface para organização e facilitar o seu uso, como ilustrado pela Figura 47.

**Figura 47 – Definição das funções do ultrassom no BIPES**

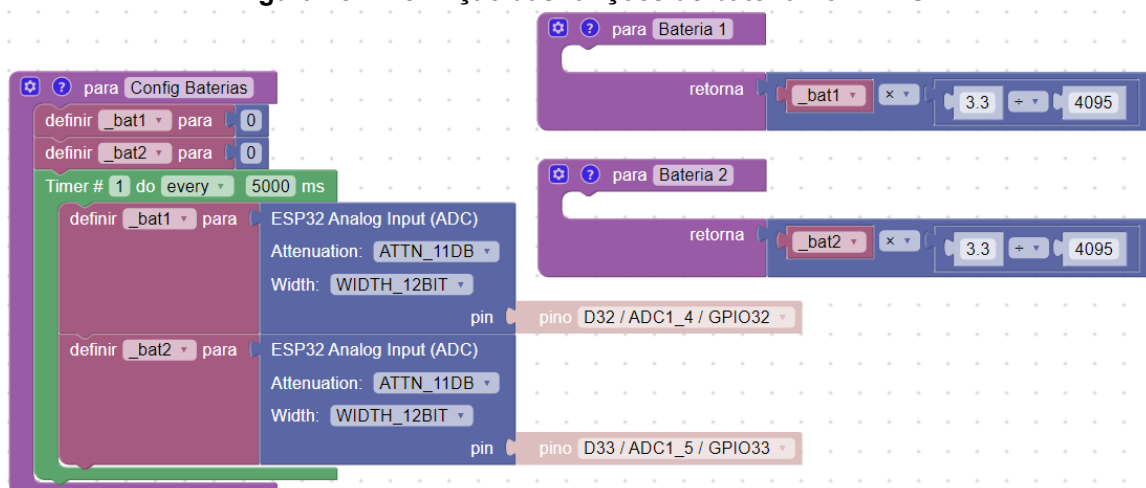


Fonte: Elaboração própria (2022).

O monitoramento das baterias é realizado através do ADC interno de 12 bits. Fazem-se leituras de 0 V até 3,3 V, a cada 5 segundo a partir do *timer*.

Implementou-se a mesma solução usada pela classe Baterias no MicroPython, convertendo o valor recebido com um fator de conversão, a Figura 48 demonstra estas funções.

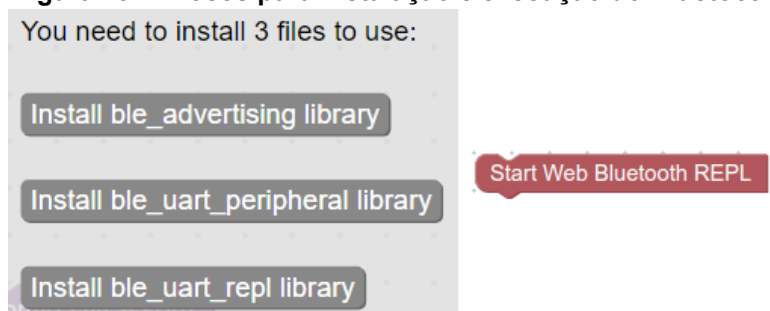
**Figura 48 – Definição das funções de bateria no BIPES**



Fonte: Elaboração própria (2022).

Ao fim da elaboração das funções para controle dos módulos do Buggy, aplicou-se a função de conexão *Bluetooth* entre a plataforma e o microcontrolador, para aferir o seu funcionamento em uma aplicação real. A conexão é realizada a partir da configuração do microcontrolador, utilizando arquivos de instalação disponíveis na plataforma, que introduzem bibliotecas de comunicação na placa e executando o bloco que dá início à comunicação, apresentados pela Figura 49.

**Figura 49 – Blocos para instalação e execução do Bluetooth**



Fonte: Elaboração própria (2022).

Após configurado e iniciado, encontra-se o dispositivo ESP 32 disponível para conexão em aparelhos com conexão *Bluetooth*. Conecta-se com o dispositivo através da plataforma utilizando a interface de conexão, demonstrada na Figura 11.

Para enviar e receber arquivos, é necessário realizar a conexão com o terminal de console, pressionando o botão “*Connect (Web Serial)*” apresentado na Figura 50.

**Figura 50 – Conexão com o terminal de console**



Fonte: Elaboração própria (2022).

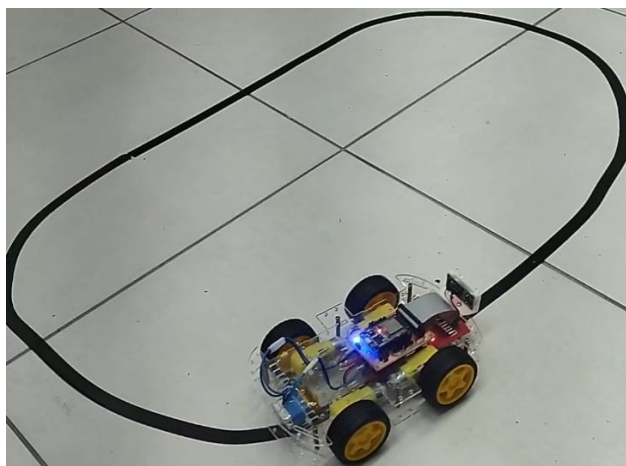
Realizada a conexão do terminal, torna-se possível enviar diretamente para o microcontrolador, através do Bluetooth, os blocos e funções concebidos anteriormente, sendo capaz de editá-los e executá-los sem a necessidade de fios.

Ao finalizar o desenvolvimento da programação em BIPES, exportou-se todas as funções criadas a partir do arquivo do projeto, pois esta plataforma não possui uma ferramenta para exportação das funções em forma de biblioteca. Assim torna-se possível armazenar e compartilhar o projeto contendo as funções necessárias para o controle do Buggy.

### **4.3 Testes das bibliotecas**

Para verificar o funcionamento das bibliotecas, *scripts* foram desenvolvidos em cada linguagem para controlar o Buggy em um circuito seguidor de linha, demonstrado na Figura 51. O objetivo desse teste é assegurar que os módulos responsáveis por movimentar e conduzir o Buggy estejam funcionando de forma adequada e mantendo-o na trilha. Tais módulos são altamente dinâmicos e dependendo da implementação do código o resultado pode ser prejudicado.

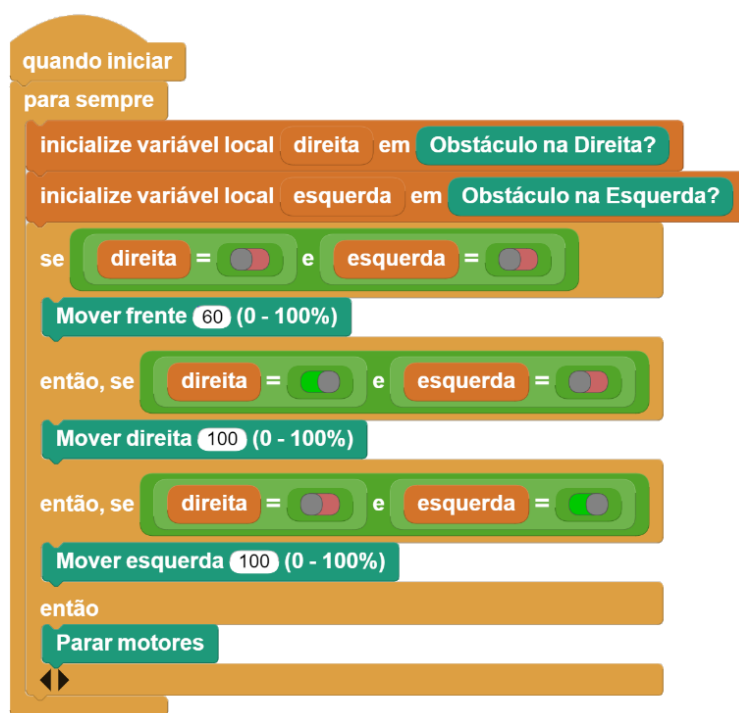
**Figura 51 – Circuito para teste do funcionamento das bibliotecas**



Fonte: Elaboração própria (2022).

A Figura 52 apresenta a implementação do código no MicroBlocks, esse *script* foi criado combinando as funções básicas de programação do MicroBlocks com a biblioteca desenvolvida durante o trabalho. Seu funcionamento se dá de maneira que o Buggy seguirá em frente até que o sensor de linha detecte um sinal positivo em um dos lados, o que fará com que o motor gire o Buggy na direção oposta e mantenha-o no trajeto.

**Figura 52 – Script seguidor de linha no MicroBlocks**

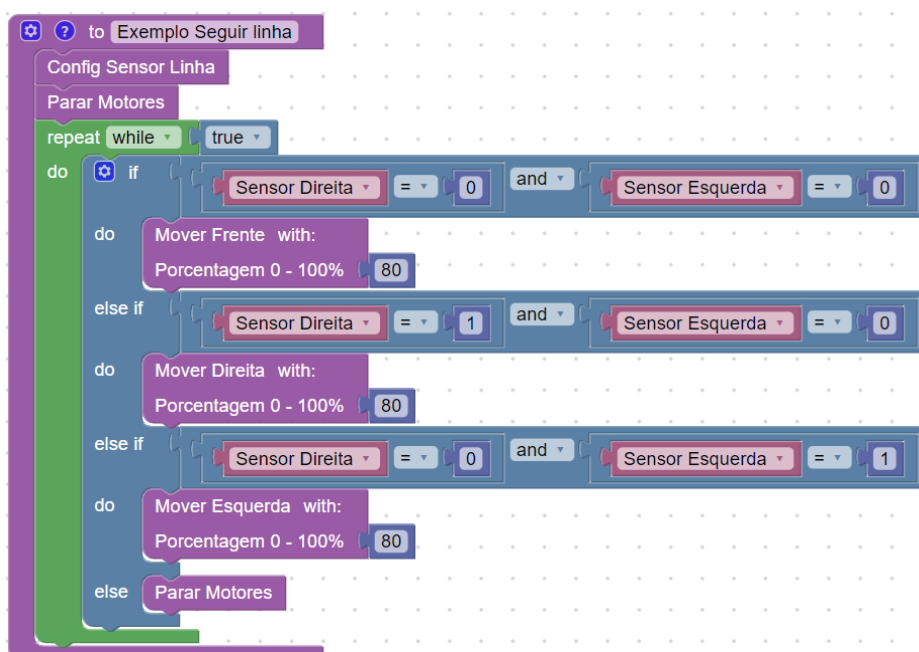


Fonte: Elaboração própria (2022).

Como a leitura dos sensores de linha no MicroBlocks é feita diretamente, sem qualquer tratamento, como é feito no MicroPython e no Bipes, foi necessário diminuir a velocidade do veículo em retas para aumentar a confiabilidade dos sensores e aumentá-la em curvas para corrigir o trajeto rapidamente. Assim, a velocidade ideal em retas foi estabelecida em 60% da velocidade máxima e em curvas 100% da velocidade máxima. Com estes valores, o circuito foi concluído com sucesso.

A Figura 53 apresenta a implementação do código no BIPES. Assim como no MicroBlocks, utilizou-se funções próprias da linguagem, combinando com a biblioteca desenvolvida durante este trabalho. Seu funcionamento se dá de maneira igual à implementada pelo MicroBlocks.

**Figura 53 – Script seguidor de linha no BIPES**



Fonte: Elaboração própria (2022).

Diferente do MicroBlocks a leitura dos sensores passa por um processo de *debounce*, reduzindo os sinais falsos positivos. Foi possível manter a velocidade do veículo em 80% da velocidade máxima em linha reta e curvas, devido à maior confiabilidade da leitura dos sensores. Com estes valores, o circuito foi concluído com sucesso e de maneira mais rápida se comparado com o MicroBlocks.

A velocidade estabelecida levou em consideração o estado de carga da bateria do Buggy, já que o torque e a velocidade máxima dos motores estão diretamente relacionados com a carga. Por isso, considerou-se uma bateria completamente carregada para estabelecer os valores de velocidade.

#### **4.4 Análise e discussão dos resultados**

Após o desenvolvimento das partes que compõem este trabalho, ou seja, a placa de interface, o desenvolvimento em MicroPython, MicroBlocks e BIPES, pode-se observar a viabilidade, limitações e dificuldades que cada plataforma de programação em blocos utilizada possui.

No quesito usabilidade, ambas as linguagens apresentam uma interface de desenvolvimento amigável e simples, com guias, referências e tutoriais para instalação, elaboração de projetos e protótipos. Focam em suas identidades visuais para facilitar o processo de programação, eliminando a necessidade de um estudo aprofundado de suas sintaxes e seus lexemas, favorecendo assim a compreensão e agilidade no desenvolvimento.

Ambas possuem uma variedade de bibliotecas integradas de sensores, atuadores e de comunicação, com blocos para sua utilização de maneira direta e sem configuração. Permite-se, assim, que os usuários criem, editem e compartilhem seus blocos e funções, facilitando a colaboração e o aprendizado entre projetistas e estudantes.

Observou-se que estas linguagens possuem limitações, principalmente por serem linguagens em blocos. Como seu princípio de funcionamento é baseado em arrastar os blocos e conectá-los, o deslocamento e edição de blocos individuais se torna uma tarefa difícil. Sua impossibilidade em implementar paradigmas de programação mais avançados como a orientação em objetos dificultam a organização e a reutilização dos códigos.

Constatou-se, como demonstra a Tabela 6, que a linguagem MicroBlocks não possui blocos para interação com interrupções e *timers*, dificultando o controle de sinais consecutivos recebidos pelo Buggy, tornando-se possível contornar este problema utilizando o seu recurso de paralelismo, distanciando-se das soluções

aplicadas em MicroPython. Pode-se afirmar também que sua incapacidade em operar com números decimais dificulta a realização de cálculos de precisão.

**Tabela 6 – Soluções implementadas para cada linguagem**

| Módulos                  | MicroPython                                                                                                                                                     | MicroBlocks                                                                                                                                                                     | BIPES                                                                                                                                                           |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Sensores de Linha</b> | <ul style="list-style-type: none"> <li>• <i>Timer</i> de 5 milissegundos.</li> <li>• <i>Debounce</i> para contagem de 10 sinais.</li> </ul>                     | <ul style="list-style-type: none"> <li>• Impossibilidade do uso de <i>timers</i>.</li> <li>• Retorna diretamente.</li> </ul>                                                    | <ul style="list-style-type: none"> <li>• <i>Timer</i> de 5 milissegundos.</li> <li>• <i>Debounce</i> para contagem de 10 sinais.</li> </ul>                     |
| <b>Motores</b>           | <ul style="list-style-type: none"> <li>• Conversão de porcentagem para <i>duty</i>.</li> <li>• Controle com PWM.</li> </ul>                                     | <ul style="list-style-type: none"> <li>• Conversão de porcentagem para <i>duty</i>.</li> <li>• Controle com PWM.</li> </ul>                                                     | <ul style="list-style-type: none"> <li>• Conversão de porcentagem para <i>duty</i>.</li> <li>• Controle com PWM.</li> </ul>                                     |
| <b>Ultrassom</b>         | <ul style="list-style-type: none"> <li>• <i>Timer</i> de 500 milissegundos.</li> <li>• Envia-se um sinal e aguarda-se a resposta.</li> </ul>                    | <ul style="list-style-type: none"> <li>• Impossibilidade do uso de <i>timers</i>.</li> <li>• Utilização de <i>pooling</i>.</li> </ul>                                           | <ul style="list-style-type: none"> <li>• Utiliza-se a biblioteca interna do sensor HCSR04.</li> </ul>                                                           |
| <b>Encoder</b>           | <ul style="list-style-type: none"> <li>• Interrupções.</li> <li>• Calcula-se a frequência de interrupções.</li> </ul>                                           | <ul style="list-style-type: none"> <li>• Impossibilidade do uso de interrupções.</li> <li>• Utiliza-se o paralelismo.</li> <li>• Retorna somente pulsos por segundo.</li> </ul> | <ul style="list-style-type: none"> <li>• Interrupções.</li> <li>• Calcula-se a frequência de interrupções.</li> </ul>                                           |
| <b>Baterias</b>          | <ul style="list-style-type: none"> <li>• <i>Timer</i> de 5 segundos.</li> <li>• Utiliza-se o ADC.</li> <li>• Necessário conversão do valor recebido.</li> </ul> | <ul style="list-style-type: none"> <li>• Impossibilidade do uso de <i>timers</i>.</li> <li>• Utiliza-se o ADC.</li> <li>• Retorno imediato</li> </ul>                           | <ul style="list-style-type: none"> <li>• <i>Timer</i> de 5 segundos.</li> <li>• Utiliza-se o ADC.</li> <li>• Necessário conversão do valor recebido.</li> </ul> |

|                           |                                     |                               |
|---------------------------|-------------------------------------|-------------------------------|
| Mesma solução e resultado | Solução diferente e mesmo resultado | Solução e resultado diferente |
|---------------------------|-------------------------------------|-------------------------------|

Fonte: Elaboração própria (2022).

As soluções implementadas para cada módulo de controle de acordo com a linguagem escolhida estão ilustradas pela Tabela 6. Observa-se que dos cinco módulos implementados pela base MicroPython, a plataforma MicroBlocks obteve sucesso apenas em uma. Isso demonstra que as limitações da plataforma fazem uma grande diferença na eficiência da execução de aplicações práticas.

No entanto, observou-se que a plataforma BIPES oferece uma integração mais conveniente e menos restritiva, combinando programação em blocos e textual em uma única ferramenta. Desta forma, foi possível implementar quatro das cinco soluções utilizadas em MicroPython. A solução implementada no controle do sensor ultrassônico, também contribuiu para a simplificação do processo de programação, pois havia um bloco pronto para uso, eliminando a necessidade de criar uma lógica de controle.

A avaliação do circuito de teste revelou uma ligeira variação na performance entre as duas linguagens. A volta mais ágil foi executada com sucesso pela plataforma BIPES, graças a sua implementação mais eficiente. Embora a aplicação em MicroBlocks não tenha alcançado resultados tão satisfatórios durante o teste, ela ainda conseguiu concluir o circuito com sucesso.

Desta forma, pode-se afirmar que o comportamento das linguagens de programação em blocos assemelhou-se ao que foi planejado através da aplicação desenvolvida com a linguagem textual MicroPython. Conclui-se com êxito, apesar de certas limitações, a aplicação em ambas as linguagens, implementando o controle de todos os módulos e sensores presentes no kit Buggy AWD.

Com isso, confirmou-se que o projeto atingiu os requisitos de estudo e desenvolvimento, habilitando o desenvolvedor e estudante de sistemas eletrônicos e embarcados a utilizar tais ferramentas para estudo e em seus futuros projetos e protótipos.

## 5 CONSIDERAÇÕES FINAIS

Neste trabalho apresentou-se a elaboração de uma aplicação embarcada desenvolvida a partir de linguagens de programação em blocos, para validar seu funcionamento e viabilidade. Com isso, projetistas e estudantes de software e hardware embarcados poderão utilizar tais ferramentas, com o propósito de facilitar a compreensão e aprendizado dos paradigmas de programação, mantendo funções semelhantes às soluções anteriores.

Após estabelecer as plataformas de programação, implementou-se a aplicação em um dispositivo real, verificando e validando seus funcionamentos individualmente por meio de testes e da comparação com a linguagem textual MicroPython. A abordagem de utilizar uma linguagem de programação textual para realizar o controle de qualidade e como referência de desenvolvimento, resultou em um projeto mais claro e consistente, facilitando a visualização das limitações e diferenças entre as linguagens baseadas em blocos.

Com o desenvolvimento do projeto finalizado, tornou-se possível testar e validar o funcionamento do Buggy AWD empregando as soluções de programação em blocos, de forma que foram alcançados os objetivos específicos estipulados na Seção 1.4.

A implementação da placa de interface para o ESP 32 e das bibliotecas de programação desenvolvidas são *open source*, tais documentos de projeto e códigos estão armazenados no GitHub do professor Renan Augusto Starke. Sendo assim, este presente projeto proporciona aos futuros e existentes alunos, uma plataforma prática para estudo de *hardware*, programação e microcontroladores.

Para trabalhos futuros sugere-se realizar a constituição de um curso introdutório para linguagens de programação em blocos focadas ao ambiente embarcado, com ênfase em conceitos básicos como variáveis, lógica condicional, lógica iterativa, estruturas e funções, utilizando exemplos práticos em kits de desenvolvimento, como o Buggy AWD. A fim de validar a eficácia deste método de programação em indivíduos inexperientes e experientes.

Pode-se também aplicar em um próximo trabalho a plataforma BIPES, juntamente com as suas vantagens na questão de comunicação sem fio e

bibliotecas próprias para internet e *Bluetooth*, para a realização de um protótipo de *hardware* IoT mais complexo, visando uma redução no desenvolvimento dos códigos, além de ser uma maneira moderna de solucionar o problema escolhido.

## REFERÊNCIAS

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. **Compilers: principles, techniques, & tools**. Pearson Education India, 2007.

BARR, Michael. **Programming embedded systems in C and C++**. O'Reilly Media, Inc., 1999.

BATES, Martin P. **Programming 8-bit PIC microcontrollers in C: with interactive hardware simulation**. Newnes, 2008.

BAU, D.; GRAY, J.; KELLEHER, C.; SHELDON, J.; TURBAK, F. **Learnable programming: blocks and beyond**. Communications of the ACM, v. 60, n. 6, p. 72-80, 2017.

BELL, Charles. **MicroPython for the Internet of Things**. Springer, 2017.

BIPES. **BIPES**. Disponível em: <https://bipes.net.br/>. Acesso em: 26 nov. 2022.

CABALLAR, Rina Diane. **Programming Without Code: the rise of no-code software development**. 2020. Disponível em: <https://spectrum.ieee.org/programming-without-code-no-code-software-development>. Acesso em: 14 out. 2022.

CABRERA, Lautaro; MALONEY, John H.; WEINTROP, David. **Programs in the Palm of your Hand: how live programming shapes children's interactions with physical computing devices**. 2019. Association For Computing Machinery, New York. Disponível em: <https://dl.acm.org/doi/10.1145/3311927.3323138>. Acesso em: 16 out. 2022.

ESPRESSIF SYSTEMS. **ESP32WROOM32**. 2022. Disponível em: [https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32\\_datash eet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datash eet_en.pdf). Acesso em: 04 nov. 2022.

GEORGE, Damien. **MicroPython-Python for microcontrollers**. 2014. Disponível em: <https://micropython.org/>. Acesso em: 14 out. 2022.

GIL, A. C. **Como elaborar projetos de pesquisa**. 4. ed. [S.l.]: Editora Atlas S.A., 2002. ISBN 9788522478408,8522478406.

GOOGLE. **Blockly**. Disponível em: <https://developers.google.com/blockly>. Acesso em: 14 out. 2022.

HOROWITZ, Ellis. **Fundamentals of programming languages**. Springer Science & Business Media, 1984.

JUNIOR, A. G. D. S.; GONÇALVES, L. M. G.; CAURIN, G. A. D. P.; TAMANAKA, G. T. B.; HERNANDES, A. C.; & AROCA, R. V. **BIPES: Block Based Integrated Platform for Embedded Systems**. IEEE Access, 2020. Disponível em: <https://ieeexplore.ieee.org/document/9246562>

KERNIGHAN, Brian W.; RITCHIE, Dennis M. **The C programming language**. Pearson Education, 1988.

KWAME, Ampomah Ernest; MARTEY, Ezekiel Mensah; CHRIS, Abilimi Gilbert. **Qualitative assessment of compiled, interpreted and hybrid programming languages**. Communications on Applied Electronics (CAE), v. 7, n. 7, p. 8-13, 2017.

MALONEY, J. RESNICK, M. RUSK, N. SILVERMAN, B. EASTMOND, E. **The scratch programming language and environment**. ACM Transactions on Computing Education, v. 10, n. 4, p. 1-15, 2010.

MALONEY, John; MÖNIG, Jens; ROMAGOSA, Bernat. **MicroBlocks**. 2021. Disponível em: <https://microblocks.fun/>. Acesso em: 26 nov. 2022.

MARCONI, M. de A.; LAKATOS, E. M. **Técnicas de Pesquisa**. São Paulo: Atlas, 2017. Citado na página 43.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY(MIT). **Scratch**. Disponível em: <https://scratch.mit.edu/>. Acesso em: 14 out. 2022.

MAZZEI, D.; MONTELISCIANI, G.; BALDI, G.; FANTONI, G. **Changing the programming paradigm for the embedded in the IoT domain**. IEEE 2nd World Forum on Internet of Things (WF-IoT). IEEE, 2015. p. 239-244.

MICHELI, Giovanni de; SAMI, Mariagiovanna. **Hardware/Software Co-Design**. Londres: Kluwer Academic Publishers, 1996.

MOORS, Luke; LUXTON-REILLY, Andrew; DENNY, Paul. **Transitioning from block-based to text-based programming languages**. 2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE). IEEE, 2018. p. 57-64.

PARDUE, Joe. **C programming for microcontrollers**. Smiley Micros. com, 2005.

RITCHIE, Dennis M. **The development of the C language**. ACM Sigplan Notices, v. 28, n. 3, 1993.

ROBOCRAZE. What is ultrasonic sensor. Disponível em: <https://robocraze.com/blogs/post/what-is-ultrasonic-sensor>. Acesso em: 04 fev. 2023.

SAMMET, Jean E. **Programming languages: History and fundamentals**. Prentice-Hall, Inc., 1969.

SEBESTA, Robert W. **Conceitos de Linguagens de Programação**. Bookman Editora, 2018.

SLONNEGER, Kenneth; KURTZ, Barry L. **Formal syntax and semantics of programming languages**. Addison-Wesley, 1995.

SPRESSIF. **Arduino ESP32**. Disponível em: <https://github.com/espressif/arduino-esp32>. Acesso em: 20 nov. 2022.

SPRESSIF. **Arduino ESP32**. Disponível em: <https://github.com/espressif/arduino-esp32>. Acesso em: 20 nov. 2022.

STARKE, Renan Augusto; Laboratório de Pesquisa Avançada em Eletrônica (LPAE). **Buggy AWD**. Disponível em: [https://github.com/xtarke/automated\\_buggy](https://github.com/xtarke/automated_buggy). Acesso em: 31 out. 2022.

STROUSTRUP, Bjarne. **The C++ programming language**. Pearson Education, 2013.

STROUSTRUP, Bjarne. **The essence of C++**. Video. Edinburgh: The University of Edinburgh, 2014.

TEJA, Ravi. **ESP32 Pinout | ESP-WROOM-32 Pinout**. Disponível em: [www.electronicshub.org/esp32-pinout/](http://www.electronicshub.org/esp32-pinout/). Acesso em: 04 nov. 2022.

TOLLERVEY, Nicholas H. **Programming with MicroPython: embedded programming with microcontrollers and Python**. O'Reilly Media, Inc., 2017.

VISHAY. **Datasheet: Reflective Optical Sensor with Transistor Output**. Electronic Publication, 2017.

WATSON, Des. **Compilers and Interpreters**. Springer, 2017.

WEINTROP, David. **Block-based programming in computer science education**. Communications of the ACM, v. 62, n. 8, p. 22-25, 2019.

WEINTROP, David; WILENSKY, Uri. **To block or not to block, that is the question: students' perceptions of blocks-based programming**. Proceedings of the 14th international conference on interaction design and children. 2015. p. 199-208.

WEINTROP, David; WILENSKY, Uri. **Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs**. Proceedings of the eleventh annual international conference on international computing education research. 2015. p. 101-110.

## **APÊNDICES**

## APÊNDICE A – Classes de controle em MicroPython

```

1. from machine import Pin, Timer
2. import time
3.
4. class SensorLinha:
5.     """Classe para controlar os sensores de linha"""
6.
7.     # Pinos dos sensores de linha
8.     R_TCRT = 26
9.     L_TCRT = 25
10.
11.    # Contagem para o debounce
12.    CONTAGEM = 10
13.
14.    def __init__(self):
15.        self.sensor_direita = Pin(self.R_TCRT, Pin.IN)
16.        self.sensor_esquerda = Pin(self.L_TCRT, Pin.IN)
17.
18.        self.estado_direita = 0
19.        self.estado_esquerda = 0
20.        self.amostras_direita = self.CONTAGEM
21.        self.amostras_esquerda = self.CONTAGEM
22.
23.        self.tim1 = Timer(1)
24.        self._init_timer()
25.
26.    def _recebe_valores(self, t):
27.        if (self.sensor_direita.value() != self.estado_direita):
28.            # Se contagem = 0, debounce terminado
29.            self.amostras_direita -= 1
30.            if (self.amostras_direita <= 0):
31.                self.estado_direita = self.sensor_direita.value()
32.        else:
33.            self.amostras_direita = self.CONTAGEM
34.
35.        if (self.sensor_esquerda.value() != self.estado_esquerda):
36.            # Se contagem = 0, debounce terminado
37.            self.amostras_esquerda -= 1
38.            if (self.amostras_esquerda <= 0):
39.                self.estado_esquerda = self.sensor_esquerda.value()
40.        else:
41.            self.amostras_esquerda = self.CONTAGEM
42.
43.    def direita(self):
44.        return self.estado_direita
45.
46.    def esquerda(self):
47.        return self.estado_esquerda
48.
49.    def _init_timer(self, period_ms = 5):

```

```

50.         self.tim1.init(period=period_ms, mode=Timer.PERIODIC,
        callback=self._recebe_valores)

```

### sensor\_linha.py

```

1. from machine import Pin, PWM
2. import time
3.
4. class Motor:
5.     """Classe para controlar os motores do buggy"""
6.
7.     # Pinos do PWM
8.     M1 = 21
9.     M1N = 22
10.    M2 = 18
11.    M2N = 19
12.
13.    # Frequencia PWM
14.    pwm_freq = 1000
15.
16.    def __init__(self):
17.        self.m1 = PWM(Pin(self.M1), self.pwm_freq)
18.        self.m1n = PWM(Pin(self.M1N), self.pwm_freq)
19.        self.m2 = PWM(Pin(self.M2), self.pwm_freq)
20.        self.m2n = PWM(Pin(self.M2N), self.pwm_freq)
21.
22.        self.parar()
23.
24.    def parar(self):
25.        self.m1.duty(0)
26.        self.m1n.duty(0)
27.        self.m2.duty(0)
28.        self.m2n.duty(0)
29.
30.    def _porcentagem_duty(self, porcentagem):
31.        return int((porcentagem * 1023) / 100)
32.
33.    def mover_frente(self, porcentagem):
34.        self.parar()
35.        if (porcentagem > 0) and (porcentagem <= 100):
36.            duty = self._porcentagem_duty(porcentagem)
37.            self.m1.duty(duty)
38.            self.m2.duty(duty)
39.
40.    def mover_re(self, porcentagem):
41.        self.parar()
42.        if (porcentagem > 0) and (porcentagem <= 100):
43.            duty = self._porcentagem_duty(porcentagem)
44.            self.m1n.duty(duty)
45.            self.m2n.duty(duty)
46.

```

```

47.     def mover_esquerda(self, porcentagem):
48.         self.parar()
49.         if (porcentagem > 0) and (porcentagem <= 100):
50.             duty = self._porcentagem_duty(porcentagem)
51.             self.m1.duty(duty)
52.             self.m2n.duty(duty)
53.
54.     def mover_direita(self, porcentagem):
55.         self.parar()
56.         if (porcentagem > 0) and (porcentagem <= 100):
57.             duty = self._porcentagem_duty(porcentagem)
58.             self.m1n.duty(duty)
59.             self.m2.duty(duty)

```

### motor.py

```

1. from machine import Pin
2. import machine
3. from time import time
4.
5. class Encoder:
6.     """Classe para controlar os Encoders de velocidade"""
7.
8.     IRQ = 20 # Num de interrupções por revolução
9.     INTERVALO = 0.5 # Frequencia que sera calculado o rpm
10.
11.     # Recebe o pino do encoder pelo construtor
12.     def __init__(self, pinEncoder):
13.         self.time0 = time() # Tempo para fazer o calculo da velocidade
14.         self.counter = 0
15.         self.rpm = 0.0
16.         self.freq = 0
17.
18.         # Definindo o pino do encoder como interrupção na borda de
19.         subida
20.         self.irq = Pin(pinEncoder, Pin.IN, Pin.PULL_UP)
21.         self.irq.irq(trigger=Pin.IRQ_RISING,
22.             handler=self.handle_interrupt)
23.
24.     def handle_interrupt(self, channel):
25.         self.counter +=1
26.
27.     def monitorRPM(self):
28.         if (time() - self.time0) > self.INTERVALO:
29.             state = machine.disable_irq()
30.             self.freq = self.counter/(time() - self.time0)
31.             self.rpm = self.freq/self.IRQ*60
32.             self.counter = 0 # Reseta o counter para a próxima revolução
33.             self.time0 = time() # Reseta o timer para próximo intervalo
34.             machine.enable_irq(state)
35.         return self.rpm

```

## encoder.py

```

1. import machine, time
2. from machine import Pin, Timer
3.
4. class Ultrassom:
5.     """Classe para controlar o ultrassom do buggy"""
6.     # Pinos do PWM
7.     TRIG0 = 27
8.     ECHO0 = 14
9.
10.    # Tempo de resposta máximo
11.    ECHO_TIMEOUT = 10000
12.
13.    def __init__(self):
14.        self.pulse_time = 0
15.
16.        self.trigger = Pin(self.TRIG0, mode=Pin.OUT, pull=None)
17.        self.trigger.value(0)
18.
19.        self.echo = Pin(self.ECHO0, mode=Pin.IN, pull=None)
20.
21.        self.tim0 = Timer(0)
22.        self._init_timer()
23.
24.    def _envia_pulso_e_espera(self, t):
25.        self.trigger.value(0) # Aguarda a estabilizacao do sensor
26.        time.sleep_us(5)
27.        self.trigger.value(1) # Envia pulso por 10us
28.        time.sleep_us(10)
29.        self.trigger.value(0)
30.        self.pulse_time = machine.time_pulse_us(self.echo, 1,
31.        self.ECHO_TIMEOUT)
32.
33.    def distancia_cm(self):
34.        # Calculo da distância é pulse_time / 2 (o pulso vai e volta)
35.        # e dividido por 29,13 porque a velocidade do som no ar é
36.        0.034320 cm/us
37.        # que é 1 cm cada 29.13us
38.        cms = (self.pulse_time / 2) / 29.1375
39.        return cms
40.
41.    def _init_timer(self, period_ms = 500): # Cria um timer para
42.        atualizar o pulse_time a cada 500ms
43.        self.tim0.init(period=period_ms, mode=Timer.PERIODIC,
44.        callback=self._envia_pulso_e_espera)

```

## ultrassom.py

```

1. from machine import Pin, Timer, ADC
2. import time
3.
4. class Baterias:
5.     """Classe para controlar as baterias"""
6.
7.     # Pinos das baterias
8.     BAT1 = 32
9.     BAT2 = 33
10.
11.     # Fator de correção do ADC
12.     FATOR_ADC = 3.3 / 4095
13.
14.     def __init__(self):
15.         self.bat1 = ADC(Pin(self.BAT1))
16.         self.bat2 = ADC(Pin(self.BAT2))
17.
18.         self.bat1_volts = 0
19.         self.bat2_volts = 0
20.
21.         # Atenuação para ler até 3.3V (Máx. do esp32)
22.         self.bat1 atten(ADC.ATTN_11DB)
23.         self.bat2 atten(ADC.ATTN_11DB)
24.
25.         self.tim2 = Timer(2)
26.         self._init_timer()
27.
28.     def _recebe_valores(self, t):
29.         self.bat1_volts = self.bat1.read()
30.         self.bat2_volts = self.bat2.read()
31.
32.     def bateria1(self):
33.         return self.bat1_volts * self.FATOR_ADC
34.
35.     def bateria2(self):
36.         return self.bat2_volts * self.FATOR_ADC
37.
38.     def _init_timer(self, period_ms = 5000): # Cria um timer para
39.         # atualizar os valores a cada 5s
40.         self.tim2.init(period=period_ms, mode=Timer.PERIODIC,
41.             callback=self._recebe_valores)

```

### bateria.py

```

1. from motor import Motor
2. from bateria import Baterias
3. from encoder import Encoder
4. from sensor_linha import SensorLinha
5. from ultrassom import Ultrassom
6. import time
7.
8. PIN_ENC1 = 15

```

```

9. PIN_ENC2 = 4
10.
11. def MostrarSensores(ultrassom, sensoresLinha, encoder1, encoder2,
    baterias):
12.     print("BAT1: {0}V BAT2: {1}V".format(baterias.bateria1(),
        baterias.bateria2()))
13.     print("rpm1: {0} rpm2: {1}".format(encoder1.monitorRPM(),
        encoder2.monitorRPM()))
14.     print("ESTADO DIREITA = ", sensoresLinha.direita())
15.     print("ESTADO ESQUERDA = ", sensoresLinha.esquerda())
16.     print("Distancia = ", ultrassom.distancia_cm())
17.
18. def ControleTeclado():
19.     motores = Motor()
20.     baterias = Baterias()
21.     encoder1 = Encoder(PIN_ENC1)
22.     encoder2 = Encoder(PIN_ENC2)
23.     sensoresLinha = SensorLinha()
24.     ultrassom = Ultrassom()
25.
26.     print('Bem vindo Ao Buggy')
27.     print('Digite a direção para mover: ')
28.     print('p: parar')
29.     print('w: frente')
30.     print('s: ré')
31.     print('a: esquerda')
32.     print('d: direita')
33.     print('q: sair')
34.
35.     motores.parar()
36.     dir = 0
37.
38.     while (dir != 'q' and dir != 'Q'):
39.         dir = input()
40.
41.         if dir == 'p' or dir == 'P':
42.             motores.parar()
43.         if dir == 'w' or dir == 'W':
44.             motores.mover_frente(80)
45.         if dir == 's' or dir == 'S':
46.             motores.mover_re(80)
47.         if dir == 'a' or dir == 'A':
48.             motores.mover_esquerda(80)
49.         if dir == 'd' or dir == 'D':
50.             motores.mover_direita(80)
51.
52.         time.sleep(1)
53.         motores.parar()
54.     MostrarSensores(ultrassom, sensoresLinha, encoder1, encoder2,
        baterias)
55.
56. def SeguirLinha():

```

```

57.     motores = Motor()
58.     baterias = Baterias()
59.     encoder1 = Encoder(PIN_ENC1)
60.     encoder2 = Encoder(PIN_ENC2)
61.     sensoresLinha = SensorLinha()
62.     ultrassom = Ultrassom()
63.
64.     print('Bem vindo Ao Buggy')
65.     print('Irei seguir a linha')
66.
67.     while True:
68.         direita = sensoresLinha.direita()
69.         esquerda = sensoresLinha.esquerda()
70.
71.         print(str(esquerda)+"-"+str(direita))
72.
73.         if direita==0 and esquerda==0:
74.             motores.mover_frente(50)
75.         elif direita==1 and esquerda==0:
76.             motores.mover_direita(80)
77.         elif direita==0 and esquerda==1:
78.             motores.mover_esquerda(80)
79.         else:
80.             motores.parar()
81.
82.     MostrarSensores(ultrassom, sensoresLinha, encoder1, encoder2,
83.                     baterias)
84. if __name__ == '__main__':
85.     #ControleTeclado()
86.     SeguirLinha()

```

buggy.py