PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA E DE ARTES
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

DESENVOLVIMENTO DE UM CARRO ROBÔ AUTÔNOMO COM SISTEMA DE TEMPO REAL

MATHEUS DIAS FERLIN MOURA

GOIÂNIA, 2023

MATHEUS DIAS FERLIN MOURA

DESENVOLVIMENTO DE UM CARRO ROBÔ AUTÔNOMO COM SISTEMA DE TEMPO REAL

Monografia apresentada ao curso de Engenharia de Computação da PUC Goiás, como requisito para conclusão do curso em Engenharia de computação.

Orientador: Prof. Dr. Sibelius Lellis Vieira.

GOIÂNIA, 2023

MATHEUS DIAS FERLIN MOURA

DESENVOLVIMENTO DE UM CARRO ROBÔ AUTÔNOMO COM SISTEMA DE TEMPO REAL

Este Trabalho de Conclusão de Curso julgado adequado para obtenção do título de Bacharel em Engenharia de Computação, e aprovado em sua forma final pela Escola de Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, em _____/_____/_____.

_____
Prof.ª Ma. Ludmilla Reis Pinheiro dos Santos
Coordenadora de Trabalho de Conclusão de Curso

Banca Examinadora:

_____
Orientador: Prof. Dr. Sibelius Lellis Vieira

_____
Prof.ª Ma. Mirian Sandra Rosa Gusmão

_____
Prof. Me. Geraldo Valeriano Ribeiro

GOIÂNIA
2023

Dedico este trabalho às pessoas que colaboraram com seu desenvolvimento, de maneira direta ou indireta e que sempre me motivaram durante essa jornada.

AGRADECIMENTOS

Agradeço a todos os professores da PUC Goiás, por me instruírem ao longo de todo o curso, e especialmente ao Prof. Dr. Sibelius Lellis Vieira, a Profa. Dra. Solange Silva, a minha família que sempre estiveram ao meu lado nos momentos de dificuldades, principalmente meus pais e minha namorada, que durante esse tempo sempre foram por porto seguro.

A estes, meu muito obrigado.

RESUMO

Este trabalho traz o estudo de um sistema de tempo real por meio de sua implementação utilizando um processador que possua a arquitetura ARM. Para isso foram realizados estudos a respeito do tema, desenvolvendo a implementação em um Kit de desenvolvimento fabricado pela MikroEletronika. Através da codificação, foi possível verificar a execução de tarefas de maneira paralela no processador, sendo controladas pelo escalonador do FreeRTOS. A relevância desse tema na área da segurança veicular e tecnologia automotiva também será abordada.

Palavras chaves: Sistema de tempo real; Arquitetura ARM; Sistema automotivo autônomo; Sensor ultrassônico, Veículo autônomo; STM32.

ABSTRACT

This work brings the study of a real-time system through its implementation using a processor that had the ARM architecture. For this, studies were carried out on the subject, developing the implementation in a development kit manufactured by MikroEletronika. Through coding, it was possible to verify the execution of tasks in parallel on the processor, being controlled by the FreeRTOS scheduler. The sacredness of this theme in the area of vehicle safety and automotive technology will also be addressed.

Keywords: Real-time system; ARM architecture; Autonomous system; Ultrasonic sensor, Autonomous vehicle; STM32.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ADC – Analogic Digital Converter

ARM – Advanced RISC Machine

CPU – Central Power Unit

SAE – Society of Automotive Engineers

ISR – Interrupt Service Routine

OMS – Organização mundial da saúde

OPAS – Organização Pan-Americana da Saúde

PWM – Pulse Width Modulation

RTOS – Real Time Operating System

SUMÁRIO

1. **INTRODUÇÃO**

Segundo a *Red Hat* (2021), veículos autônomos são todos aqueles capazes de utilizar sensores, para observarem o ambiente ao seu redor, buscando tomar uma decisão de qual direção seguir. A partir disso, a *Society of Automotive Engineers* (SAE), organização responsável por estabelecer normas e padrões a serem seguidos em toda indústria automobilística, subdividiu o nível de automação de veículos em 6 categorias diferentes. Tal divisão pode ser resumida em dois grupos: com assistência humana e direção automatizada. A primeira, busca por meio de automações, auxiliar na direção realizada por uma pessoa. A segunda, busca implementar um veículo capaz de tomar decisões por conta própria. (SAE, 2021)

Segundo a Organização Pan-Americana da Saúde (OPAS), os erros humanos são os principais fatores de riscos para as mortes em acidentes de trânsito. Além disso, a instituição aponta que 93% das mortes ocorrem em países de baixa e média renda, embora tais países concentrem aproximadamente 60% do número de veículos do mundo. A partir disso, a Organização Mundial da Saúde (OMS), lançou em 2017 um pacote técnico para a segurança no trânsito, o Salvar VIDAS (OPAS, 2018)

Com isso, a OMS estabeleceu algumas recomendações a serem seguidas pelos fabricantes, tais como a implementação de novas tecnologias, como sistema de adaptação de velocidade, controle eletrônico de estabilidade, freios ABS e controle de tração. Dessa forma, busca-se ofertar até 2030, o acesso a sistemas de transportes seguros, sustentáveis e de preço acessível para todos. (OMS, 2017)

Segundo a *Red Hat* (2022), processadores *Advanced RISC Machine* (ARM) são uma família de *central processing units* (CPUs) baseados em um conjunto reduzidos de instruções. Além disso, Processadores ARM são desenvolvidos com foco em oferecer um design com menor tamanho, menor consumo de energia, alta velocidade de processamento e maior duração de bateria. Buscando trazer todos esses atributos pelo menor custo possível.

Segundo a *Wind River,* um sistema operacional de tempo real, em inglês (RTOS) é um sistema que possui duas funções chaves: previsibilidade e determinismo. Em um RTOS, várias tarefas são reproduzidas dentro de um curto

espaço de tempo. Além disso, em um RTOS, deve-se ser possível saber quanto tempo uma tarefa demorará para ser executada e que produzirá o mesmo resultado sempre (*Wind River*, s.d)

Um RTOS pode ser classificado de três maneiras: *Hard*, *Soft* e *Firm*. No primeiro, o sistema deve garantir que a tarefa será executada em determinado período. No segundo, apesar de possuir um tempo limite, o sistema é mais adaptável a variações na duração das tarefas. No terceiro, o RTOS deve seguir os prazos limites, porém caso não seja seguido, não possuirá grande impacto. (*Geeks for Geeks*, 2022)

Segundo a *Geeks for Geeks* (2022), RTOSs podem ser encontrados em sistemas de controles aéreos, sistemas de controle de comando, carros autônomos e serviços de rede multimídia. Justifica-se estudar esse tema, pois é necessário existir um maior nível de autonomia nos veículos, para que seja possível diminuir o fator de erro humano em acidentes (OPAS, 2018).

Diante deste contexto, este projeto visa responder a seguinte questão: **- É possível implementar um software de tempo real em um carrinho robô utilizando a arquitetura ARM?**

1.1 Objetivos

1.1.1Geral

- Desenvolver um carro em miniatura que implemente um sistema de tempo real, capaz de executar múltiplas tarefas simultaneamente, utilizando um processador ARM.

1.1.2 Específico

- Implementar um sistema de tempo real.
- Implementar sensores que identifiquem obstáculos.

1.2 Organização do trabalho

Este trabalho está dividido da seguinte forma: o capítulo 2 discorrerá sobre uma revisão bibliográfica dos assuntos utilizados para o desenvolvimento do problema proposto, mostrando os caminhos percorridos e o conteúdo estudado para que fosse possível sua conclusão. O capítulo 3 trata da estrutura do projeto de pesquisa e de suas diferentes formas e características. O capítulo 4 apresenta o desenvolvimento do trabalho, suas partes, as soluções propostas, tanto em código quanto em hardware e o detalhamento de cada ponto desenvolvido. Por fim, no capítulo 5 serão expostas as conclusões obtidas a partir dos experimentos realizados, além de possíveis melhorias para trabalhos a serem desenvolvidos futuramente.

2. Referencial Teórico

Neste capítulo é realizada uma abordagem sobre os conceitos necessários para o desenvolvimento do projeto.

2.1 Motores

Segundo Braunl (2022), existem diversas formas em que atuadores robóticos podem ser construídos. Motores elétricos ou atuadores pneumáticos com eletroválvulas são os mais comumente utilizados.

Motores elétricos podem ser encontrado com características diferentes, entre elas: corrente direta (DC) ou corrente alternada (AC), escovado ou não escovado, monofásico ou trifásico. Sendo os motores DC do tipo monofásicos os mais comumente utilizados para a locomoção de robôs. Além disso, são mais limpos, menos barulhentos e possuem a capacidade de produzir potência suficiente para o desenvolvimento de várias tarefas. (BRAUNL, 2022)

Na figura 1 pode ser visto um modelo de motor de corrente contínua. Esse motor depende de uma entrada de tensão, representado pela variável Va, que é aplicada em seus terminais, o que gerará uma corrente no corpo do motor. Seu torque será proporcional a essa corrente, ou seja, quanto maior a corrente, maior o torque. (BRAUNL, 2022)

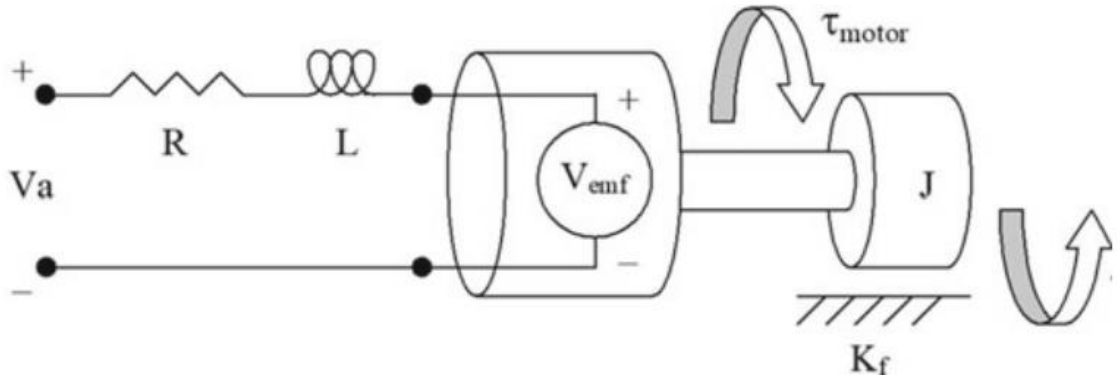

Figura 1: Modelo de motor DC (BRAUNL, 2022)

O sistema elétrico do motor pode ser modelado por um par de resistor-indutor conectados em série, representados pelas variáveis R e L respectivamente na figura 1, com uma voltagem dada pela força eletromotriz inversa, representada por Vemf na figura 1. Essa voltagem é produzida pelo movimento das bobinas do motor através de um campo magnético. (BRAUNL, 2022)

2.1.1 Ponte H

Em uma aplicação que existe a necessidade de alterar a direção e a velocidade de rotação do motor, se faz necessário a utilização de uma ponte H. Esse recurso possibilita alterar as polaridades das entradas, fazendo com o que o motor gire em sentido horário ou anti-horário.

Na figura 2, podemos ver um motor com dois terminais *a* e *b*. Uma fonte de alimentação, com os polos positivo (+) e negativo (-). Ao fechar as chaves 1 e 2, conectaremos *a* com o polo positivo e *b* com o polo negativo. Fazendo com o que o motor gire para frente. Ao abrir 1 e 2 novamente e agora fechando as chaves 3 e 4, conectamos *a* com o polo negativo e *b* com o polo positivo. Fazendo com o que o motor gire para trás. (BRAUNL, 2022)

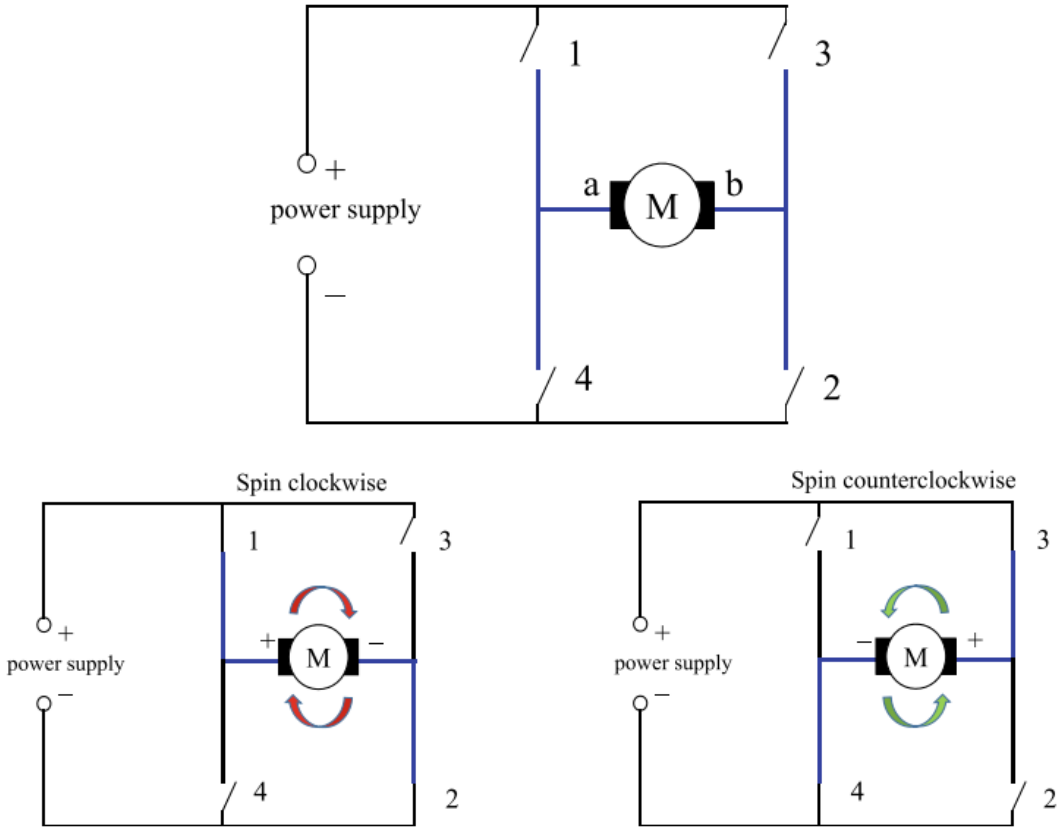

Figura 2: Operação da ponte H (BRAUNL, 2022)

2.1.2 *Pulse Width Modulation* (PWM)

O pulso com modulação de comprimento, em inglês, PWM é um método para evitar a utilização de um circuito analógico de energia, utilizando do fato de que sistemas mecânicos possuem uma latência relativamente alta. Ao invés de gerar um sinal analógico de saída com a tensão proporcional a velocidade desejada do motor, são gerados impulsos digitais na tensão máxima do sistema. Ao variar o comprimento das ondas utilizando o recurso de software, também se altera a saída analógica equivalente, dessa forma, controlamos a velocidade do motor, como pode ser visto na figura 3. (BRAUNL, 2022)

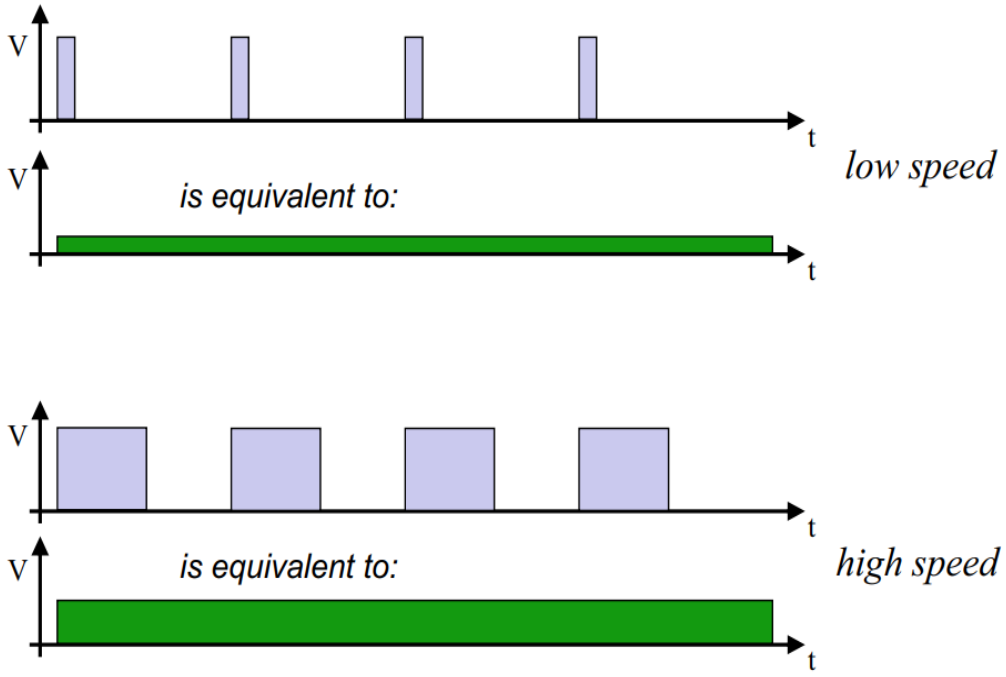

Figura 3: Funcionamento do PWM (BRAUNL, 2022)

Segundo Braunl (2022), o sinal do PWM pode ser gerado através do software, onde vários microcontroladores já possuem modos especiais e portas de saída específicas para essa função.

2.2 Sensores

Os sensores podem ser classificados conforme seus sinais de saída, sendo eles: sinal binário, sinal analógico, sinal de temporização, link serial, link paralelo. Também podem ser classificados com base em seu local de operação, sendo eles internos, como: sensor de bateria, sensor de temperatura do processador, acelerômetros, giroscópios, bússolas etc. Ou externos, como: Câmeras, sonares, radares, sensores infravermelhos, sensores ultrassônicos etc. Além disso, também pode-se classificar por suas funções, sendo elas ativas, onde o sensor monitora o ambiente sem interagir com ele, ou ativa, onde os sensores interagem diretamente com o ambiente ao seu redor, como no caso de sensores infravermelhos. (BRAUNL, 2022)

2.2.1 Sensores Ultrassônicos

Sensores ultrassônicos utilizam do princípio de disparar uma curta onda sonora, em torno de 1ms, em uma frequência ultrassônica entre 50kHz e 250kHz, medindo o tempo entre a emissão do sinal e a recepção do seu eco no sensor. Essa apuração é proporcional ao dobro da distância do objeto mais próximo ao sensor. O comportamento pode ser visto na figura 3. (BRAUNL, 2022)

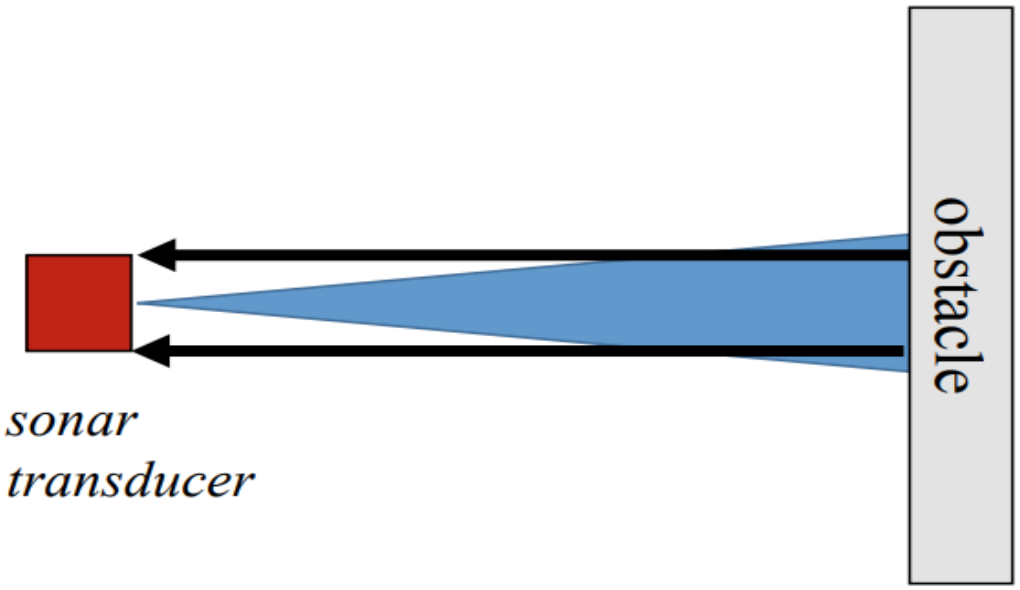

Figura 4: Funcionamento sensor ultrassônico (BRAUNL, 2022)

Segundo Wei (2015, apud KINSLER, et al., 2000), após o sensor emitir o feixe de ondas sonoras, a distância pode ser determinada a partir do tempo em que o som demora para retornar ao ponto de origem, para isso, é necessário entender a velocidade do som. Considerando que o meio transitado seja o ar, a velocidade do som pode ser calculada pela seguinte fórmula: $v = \sqrt{y * P \div D}$, onde v é a velocidade do som, y é a razão entre calores específicos do meio, sob pressão e volume constantes, P é a pressão do meio e D é a densidade do meio. Dessa forma, para uma situação de 0 graus celsius, ao nível do mar, a velocidade do som pode ser considerada 331,5 m/s. Na tabela 1 podemos ver a velocidade do som calculada para algumas temperaturas.

Temperatura em °C	Velocidade do som em m/s
+35	352,1
+30	349.2
+25	346.3
+20	343.4
+15	340.4
+10	337.5
+5	334.5
0	331.5
−5	328.4
−10	325.3
−15	322,3
−20	319,1
−25	315,9

Tabela 1 – Velocidade do Som no ar (WEI)

Para calcular a distância em metros de um objeto, é necessário utilizar a seguinte fórmula: Distância = Tempo do Echo do sinal * Velocidade do som / 2. Para o cálculo em centímetros, considerando um ambiente em que a temperatura seja 20 graus celsius positivos, temos que Distância = Echo do sinal (ms) * 0.0343 / 2.

Apesar de ser um sistema sensorial potente, existem alguns fatores que podem atrapalhar o funcionamento adequado e a confiabilidade das medidas obtidas, como por exemplo uma situação em que o sinal é refletido em uma parede angulada, fazendo com que o obstáculo pareça estar mais longe do que a parede que refletiu o sinal. Além disso, quando são utilizados vários sensores ultrassônicos em um mesmo robô, pode haver interferências entre eles, onde um sensor pode capturar a onda emitida pelo outro, essa situação pode gerar dados imprecisos, fazendo com que as distâncias não sejam reais. (BRAUNL, 2022).

2.3 Microcontroladores e microprocessadores.

Um microcontrolador pode ser definido como um computador com chip único. Analisando a morfologia da palavra, vemos que Micro se refere ao tamanho do componente, ou seja, que é bem pequeno. Controlador sugere que o dispositivo possa ser utilizado para aplicações de controle. Um microcontrolador também pode ser compreendido como um controlador embarcado, já que, em sua maioria, estão acoplados aos sistemas em que controlam. (IBRAHIM, 2020)

Um microprocessador se difere de um microcontrolador em várias formas. A principal delas é que o microprocessador requer vários outros componentes para o seu funcionamento, como: memória de programa, memória de dados relógio do circuito, sistema de interrupção etc. (IBRAHIM, 2020)

Um microcontrolador costumar ser utilizado para realizar apenas uma tarefa, seja ela controlar um motor, ligar um LED, controlar a temperatura etc. Basicamente, um microcontrolador executa um programa carregado em sua memória, recebendo dados de dispositivos externos e enviando para outros dispositivos externos. Por exemplo, em uma aplicação baseada em microcontrolador, a leitura da temperatura é realizada por um sensor de temperatura, enviada para o microcontrolador que por sua vez envia um sinal para o ativar ou desativar uma ventoinha, como pode ser visto na figura 4. (IBRAHIM, 2020)

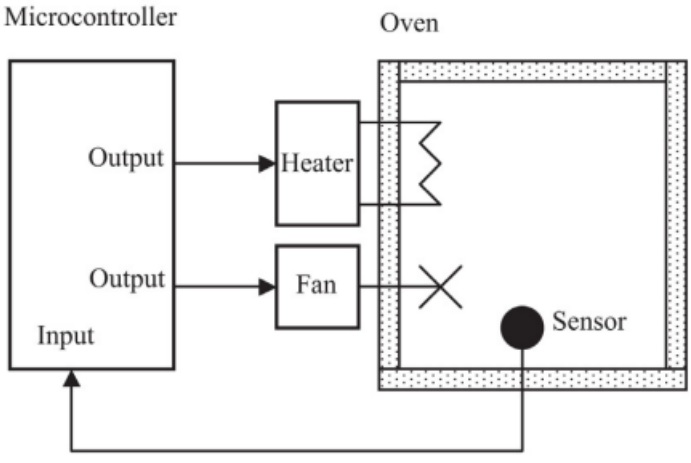

Figura 5: funcionamento de um microcontrolador (IBRAHIM, 2020)

2.3.1 Recursos dos microcontroladores

Esse tópico trata sobre alguns recursos que estão presentes nos microcontroladores e suas características.

2.3.1.1 Tensão de Alimentação

A maioria dos microcontroladores funciona com uma tensão de +5V ou 3.3V. Um regulador de tensão pode ser utilizado para se obter o valor esperado da fonte de alimentação. Se um microcontrolador utiliza uma fonte de alimentação de 9V e possui uma tensão de 3.3V de funcionamento, o regulador torna-se obrigatório. (IBRAHIM, 2020)

2.3.1.2 Relógio

Todos os microcontroladores necessitam de um relógio, ou oscilador para seu funcionamento. O relógio, em inglês *clock,* costuma ser criado por cristais e capacitores externos. Alguns microcontroladores possuem circuitos temporizadores construídos internamente e não necessitam de um mecanismo externo. O relógio interno de um microcontrolador costuma ser suficiente para maioria das aplicações, porém, em situações que requerem uma medição exata dos ciclos de *clock*, é recomendado a utilização de um circuito externo. (IBRAHIM, 2020)

2.3.1.3 Temporizadores

Temporizadores, em inglês *timers*, são partes essenciais de qualquer microcontrolador. Um timer é em sua essência um contador acionado por um pulso de *clock* externo ou pelo oscilador interno do microcontrolador. Um temporizador pode ser de 8, 16 ou 32 bits. Os dados podem ser controlados utilizando recursos de software, podendo ser utilizados para gerar sinais de interrupção. Essas interrupções podem ser utilizadas para que o programa consiga realizar operações com um tempo preciso, evitando tempo ocioso no controlador. (IBRAHIM, 2020)

2.3.1.4 Interrupções

As interrupções são recursos que fazem um microcontrolador responder rapidamente para eventos externos e internos. Quando ocorre uma

interrupção, o microcontrolador interrompe seu fluxo normal de execução, pulando para uma parte específica do programa, essa parte é chamada em inglês de *interrupt service routine* (ISR), em tradução livre, rotina do serviço de interrupção. O código de programa dentro do ISR é executado, após isso o código volta para seu fluxo normal de execução. (IBRAHIM, 2020)

Um microcontrolador pode possuir várias fontes de interrupções diferentes, sendo possível definir diferentes níveis de prioridade entre elas. Dessa forma, uma interrupção com alta prioridade pode tomar o lugar no processador de uma com baixa prioridade. (IBRAHIM, 2020)

2.3.1.5 Conversor analógico-digital

Um conversor analógico-digital (ADC) é utilizado para converter um sinal de uma fonte analógica, como a tensão, para um formato digital, para que essa informação possar ser lida e processada no microcontrolador. Hoje em dia, a maioria dos microcontroladores possuem um ADC construído internamente. O processo de conversão de analógico para digital deve ser iniciado pelo código do programa e pode demorar vários microssegundos para uma conversão ser completada. O ADC costuma gerar uma interrupção quando a conversão está completa, para que o programa consiga ler o valor da forma mais rápida possível. Essa forma de conversão é essencial, pois a maioria dos sensores possuem uma saída analógica. (IBRAHIM, 2020)

2.3.2 Arquitetura dos microcontroladores

Normalmente são utilizados dois tipos de arquiteturas nos microcontroladores, a arquitetura de *Harvard* e a arquitetura de *Von Neumann.*
A maioria dos microcontroladores costumam utilizar a arquitetura de *Von Neumann*, onde instruções e dados utilizam o mesmo barramento e todo o espaço de memória também está no mesmo barramento. Na arquitetura *Harvard*, dados e códigos estão em barramentos separados, como pode ser visto na figura 5. (IBRAHIM, 2020)

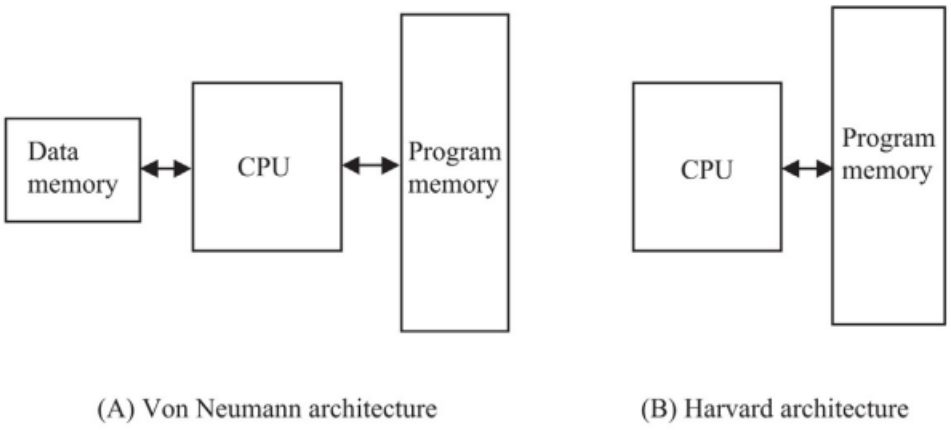

Figura 6: arquitetura de Von Neuman e arquitetura de Harvard **(IBRAHIM, 2020)**

2.3.2.1 Arquitetura CISC e RISC

Tanto RISC quanto CISC referem-se à complexidade das instruções do microcontrolador, sendo o primeiro um conjunto de instruções reduzidas e o segundo um conjunto de instruções complexos. Em um processador RISC, os dados possuem 8 bits e as instruções ocupam uma *word* na memória do programa. Dessa forma, as instruções podem ser buscadas e processadas em um mesmo ciclo de relógio, o que faz com que tenha um ganho de performance. O conjunto de instruções RISC é encontrado em processadores com a arquitetura ARM. (IBRAHIM, 2020)

2.4 Arquitetura ARM

A arquitetura ARM foi criada inicialmente na década de 1980 pela *Acorn Computers*, para serem utilizados em seus computadores próprios. Com o tempo, cada vez mais produtos foram utilizando de processadores com a arquitetura ARM. No ano de 2010, 95% dos smartphones, 10% dos computadores móveis e 35% das televisões inteligentes estavam utilizando a tecnologia ARM. (IBRAHIM, 2020)

O principal diferencial da arquitetura ARM é seu baixo consumo de energia, o que o torna ideal para aplicações que utilizem de bateria. Nos dias atuais, quase todos os aparelhos celulares possuem um chip ARM. (IBRAHIM, 2020)

Os processadores ARM são baseados em um conjunto de instruções chamados Thumb. Esse conjunto consegue obter instruções de 32-bits e comprimi-las para 16-bits, reduzindo assim o tamanho do hardware e

consequentemente seu custo de fabricação. Além disso, é utilizada uma arquitetura complexa de pipelines com vários estágios para aumentar a taxa de transferência de maneira significativa. (IBRAHIM, 2020)

Processadores ARM utilizam de uma arquitetura RISC, onde apenas uma pequena quantidade de instruções é utilizada. Resultando em um ganho de velocidade, uma vez que as instruções não tão importantes e pouco utilizadas são removidas e os caminhos dos dados são aprimorados para garantir uma melhor performance. (IBRAHIM, 2020)

2.5 Arquitetura do STM32F407VCT6

Segundo Ibrahim (2020), o microcontrolador STM32F407VCT6 é baseado na arquitetura do Cortex-4, dessa forma, possui algumas características básicas, como:

- Arquitetura 32-bits RISC
- Unidade de ponto flutuante (FPU) e processador de sinal digital (DSP)
- 168Mhz de frequência de operação máxima
- Até 1Mb de memória flash
- Até 192Kb de SRAM
- Alimentação de tensão entre 1.8 e 3.6V
- Temperatura de operação entre -40 e 105 graus celsius
- *Clock* PLL
- Até 17 timers diferentes
- Sensor de temperatura
- 2 interfaces USB
- Até 140 portas de entrada e saída
- 2 timers de Watchdog
- 2 controladores de 16bits para PWM
- Gerador de número aleatório
- Interface SDIO

Na figura 6 podemos ver um resumo das características desse microcontrolador.

Figura 7: Características básicas do microcontrolador STM32F407VCT6 **(IBRAHIM, 2020)**

2.5.1 Portas de Entradas e Saídas de uso gerais

Segundo Ibrahim (2020), as portas de entrada e saída de uso geral (GPIO) são divididas em 5. Cada uma delas de 16 bits e nomeadas A, B, C, D e E. Cada uma das portas possuí seu próprio *clock,* além das seguintes características básicas:

- A maioria dos pinos suportam +5V quando utilizados como entradas
- A velocidade de cada porta pode ser configurada no software
- Os pinos podem ser Entrada e Saída (I/O) digitais ou terem funções alternativas (DAC, SPI, USB, PWM)
- Cada pino pode ser utilizado com uma das 15 funções alternativas (AF)
- Manipulação de bits pode ser utilizada em cada pino.

Segundo Ibrahim (2020), cada porta de I/O possui os seguintes registradores, que podem ser programados utilizando o software mikroC Pro for ARM

- 2 - 32-Bit registradores de configuração GPIOx_CRL e GPIOx_CRH
- 2 - 32-Bit registradores de dados GPIOx_IDR e GPIOx_ODR
- 1 - 32-Bit registrador de definição/redefinição GPIOx_BSRR
- 1 - 16-bit registrador de redefinição GPIOx_BRR
- 1 - 32-bit registrador de bloqueio GPIOx_LCKR

2.6 Sistemas operacionais de tempo real (RTOS)

Em princípio, existem dois tipos de sistemas operacionais de tempo real (RTOS), que podem ser definidos como *hard* e *soft.* Um RTOS do tipo *hard* deve obrigatoriamente conseguir atender os deadlines de execução, enquanto em um RTOS *soft* é aceitável que alguns deadlines sejam perdidos. (HEE et al. 2019)

Dessa forma, RTOS do tipo *hard* são essenciais para sistemas de tempo reais críticos, que geralmente são aqueles encontrados nas indústrias automotivas e militares, onde perder um prazo de execução pode resultar em alguma situação desastrosa. Por exemplo, se o prazo para um airbag ser ativado após uma colisão é de 50ms, o sistema obrigatoriamente deve ser capaz de responder antes desse deadline. (HEE et al. 2019)

A essência de um RTOS é chamada de kernel ou *scheduler*, em português, escalonador. Ele é responsável por gerenciar os períodos de execução das tarefas dentro do sistema. Em um RTOS, podemos ter várias tarefas sendo executadas ao mesmo tempo, utilizando algumas técnicas, dando a impressão de que possuímos uma execução em paralelo, mesmo utilizando apenas um núcleo de processamento, podendo ser chamado de *multitasking*, ou multitarefas. (HEE et al. 2019)

Além disso, cada tarefa pode possuir um nível de prioridade diferente no sistema, onde o Kernel é responsável por garantir que ela seja sempre executada antes das demais. Dessa forma, o escalonador é responsável por interromper a execução de uma tarefa com menor prioridade, para dar lugar ao processamento de uma tarefa com maior prioridade. (HEE et al. 2019)

2.7 FreeRTOS

Dentro do freeRTOS, cada *thread* de execução pode ser chamado de tarefa, ou em inglês, *task*. Essa tasks são implementadas no formato de funções utilizando a linguagem C, a sua única característica especial é em sua assinatura, onde deve sempre retornar um tipo vazio, em inglês *void* e receber como parâmetro de entrada um ponteiro do tipo void. (BARRY, 2009)

Cada tarefa pode ser considerada como um pequeno programa por si só, tem um ponto de entrada, onde vai sempre estar rodando dentro de um loop infinito e não tem um ponto de saída. Tasks dentro do FreeRTOS não podem retornar dentro de sua implementação, ou seja, não podem ter uma cláusula do tipo *return.* A definição de uma tarefa pode ser utilizada para a criação de várias outras, sendo elas a execução dessa mesma tarefa em uma instância separada. O formato de uma tarefa pode ser visto na figura 7 (BARRY, 2009)

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance
of a task created using this function will have its own copy of the
iVariableExample variable.  This would not be true if the variable was
declared static - in which case only one copy of the variable would exist
and this copy would be shared by each created instance of the task. */
int iVariableExample = 0;

    /* A task will normally be implemented as in infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop
    then the task must be deleted before reaching the end of this function.
    The NULL parameter passed to the vTaskDelete() function indicates that
    the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

Figura 8 – Exemplo de uma tarefa no FreeRTOS (BARRY, 2009)

2.8 Trabalhos Correlatos

Existem vários trabalhos que utilizam de sensores e sistemas de tempo real para a automação de veículos. Wei (2015), apresenta diversos métodos de se evitar obstáculos em veículos autônomos, em seu trabalho exemplifica vários tipos de sensores e algoritmos para identificar qual a forma mais apropriada de implementar a identificação de objetos em veículos autônomos.

Garcia (2015) mostra a utilização de um sistema de controle de cruzeiro adaptável, por meio de simulação computacional e análise de desempenho, desenvolvendo um protótipo para validar os números obtidos no experimento e validar sua efetividade.

3. MATERIAIS E MÉTODOS

Esta pesquisa, segundo sua natureza é um resumo de assunto, onde busca explicar a área de conhecimento do projeto, buscando analisar e correlacionar as áreas de conhecimento do projeto, levando à compreensão de suas causas e explicações (WAZLAWICK, 2014)

Segundo os objetivos, possui uma parte exploratória e uma parte descritiva. Em sua parte descritiva, busca dados mais consistentes sobre o assunto, porém, não ocorre a interferência do pesquisador, buscando apenas expor os fatos da maneira que eles são (WAZLAWICK, 2014).

A pesquisa, na sua parte exploratória, pode ser considerada como a primeira parte do processo de pesquisa, por não necessariamente o autor possui um objetivo ou hipótese definida (WAZLAWICK, 2014). Essa abordagem tem como objetivo facilitar a criação de novas hipóteses. Pode ser considerada uma pesquisa flexível, pois considera aspectos variados referentes aos fenômenos ou fatos estudados (GIL, 2017)

Quanto aos procedimentos técnicos, será uma pesquisa bibliográfica e experimental. Uma pesquisa bibliográfica necessita do estudo de artigos, teses, livros, entre outras fontes. Uma pesquisa experimental se caracteriza por ter variáveis experimentais que o pesquisador tem o poder de coordenar (WAZLAWICK, 2014).

A pesquisa experimental consiste em provocar mudanças no ambiente de atuação, onde será analisado os resultados e verificar se estes são condizentes com o que era esperado (WAZLAWICK, 2014). Além disso, segundo Gil (2017), uma pesquisa experimental deve realizar pelo menos um dos experimentos que julga ser responsável por gerar a situação que está sendo pesquisada.

Quanto à abordagem, a pesquisa é do tipo quantitativa, pois utiliza métodos que buscam empregar medidas padronizadas e sistemáticas, buscando facilitar a comparação entre a análise de medidas estatísticas dos dados (AUGUSTO, 2012)

De acordo com os meios utilizados para investigação, inicia-se com uma pesquisa bibliográfica e experimental, procurando utilizar de livros, revistas, dissertações e artigos acadêmicos relacionados ao tema: automação automotiva, sistemas de tempos real e automação.

Inicialmente, será realizada uma revisão dos conceitos de sistemas de tempo real, utilizando de fontes confiáveis e estabelecidas. Posteriormente, será realizada um estudo em técnicas de otimização de performance, utilizando a linguagem C e Assembly.

Os experimentos serão realizados em ambiente controlado, utilizando de uma superfície plana, com boa aderência, seca e sem irregularidades. O sistema utilizado para a implementação do código será um Windows 11, com a seguinte configuração: Ryzen 9 5900x, RTX 3080 12GB, 1TB SSD M.2, 32 GB RAM DDR4 3600Mhz. O ambiente de desenvolvimento integrado (IDE) escolhido para a implementação do código é o Mikro C PRO for ARM. Também será utilizado um kit de desenvolvimento da MikroEletronika chamado *Buggy*.

4. DESENVOLVIMENTO PRÁTICO

O projeto, em sua parte experimental, possui como objetivo criar um modelo, em escala reduzida, com componentes acessíveis, para simular um sistema de tempo real em um microcontrolador com arquitetura ARM, considerando suas dificuldades e limitações. Com um funcionamento satisfatório em um modelo reduzido, é possível demonstrar que o código utilizado no projeto, juntamente a capacidade do microcontrolador, tornaria possível sua utilização em uma aplicação automotiva.

4.1 Componentes utilizados

Para a execução desse trabalho, foi escolhido o projeto *Buggy* disponibilizado pela empresa MikroEletronika, como pode ser visto na figura 8. Além desse projeto base, foram utilizados componentes eletrônicos que podem ser encontrados em lojas especializadas no assunto, como no caso dos cabos e sensor ultrassônico.

Figura 9– Buggy MikroEletronika. (https://www.mikroe.com/buggy)

Segundo a MikroEletronika, o Buggy é uma plataforma robótica, com possibilidade de expansão para diversos sensores e transceptores, devido a utilização do soquete mikroBUSTM. O projeto contém: 2x painéis laterais, 3x

placas mikroBUS, 4x rodas removíveis, 1x bateria do tipo LiPo e 1x placa de circuito com 4 motores integrados e um controlador ARM STM32. O projeto vem desmontado, conforme figura 9. Além desses atributos, o Buggy também conta com alguns LEDs integrado em sua placa, sendo eles: 2x LEDs frontais na cor amarela, 2x LEDs traseiros na cor amarela, 2x LEDs frontais na cor branca e 2x LEDs traseiros na cor vermelha.

Ressalta-se que os pinos em que os LEDs estão atribuídos são fixos e definidos pela MikroEletronika. A operação dos LEDs amarelos do lado esquerdo é realizada de maneira conjunta, assim como os do lado direito. Os LEDs brancos também possuem uma operação conjunta, assim como os LEDs vermelhos.

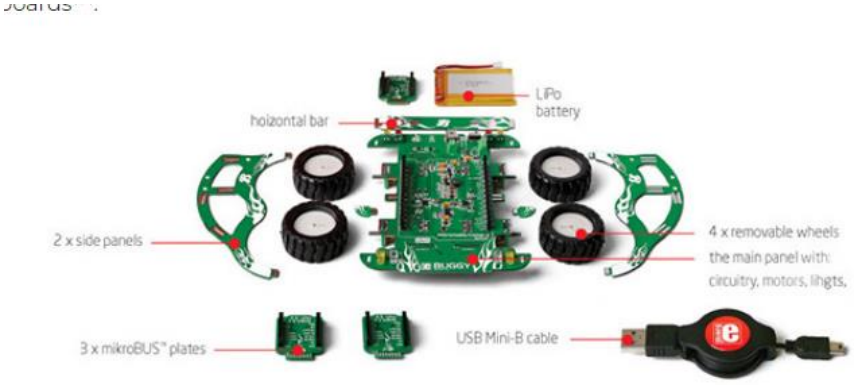

Figura 10: Componentes do projeto Buggy (https://www.mikroe.com/buggy)

O Buggy possui 108.4mm de largura, 139mm de comprimento, 61mm de altura na parte dianteira e 84mm de altura na parte traseira, conforme pode ser visto na figura 10.

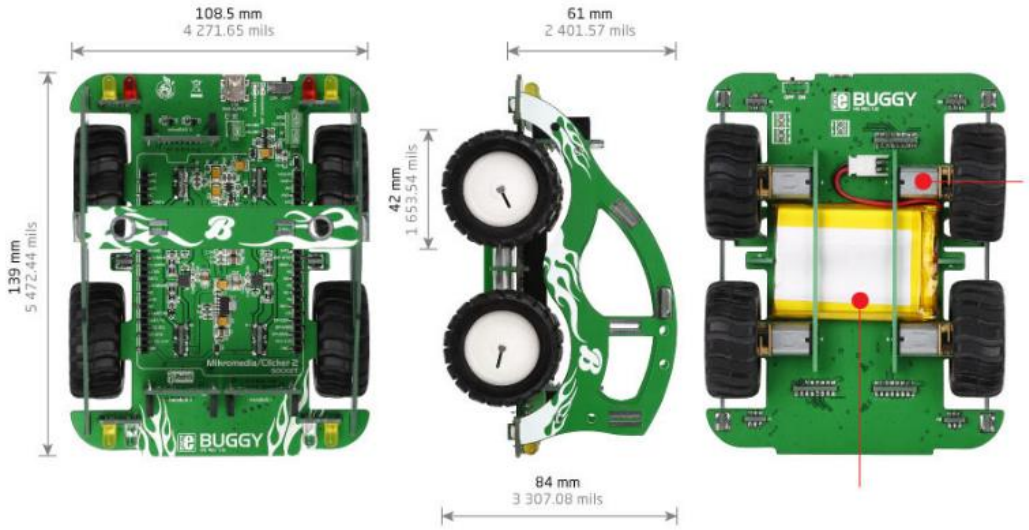

Figura 11: Dimensões do Buggy. (https://www.mikroe.com/buggy)

Além do projeto do Buggy, foram utilizados um sensor ultrassônico HC-SR04, que pode ser visto na figura 11. Também foram utilizados cabos e uma pequena protoboard para realizar as ligações entre o sensor e a placa controladora.

Figura 12: Sensor Ultrassônico. (https://www.eletrogate.com/)

Para a implementação do sensor ultrassônico, implementa-se um divisor de tensão, assim, foram utilizados dois resistores, sendo eles de 1K Ohms e 2K Ohms. O circuito do divisor de tensão pode ser visto na figura 12.

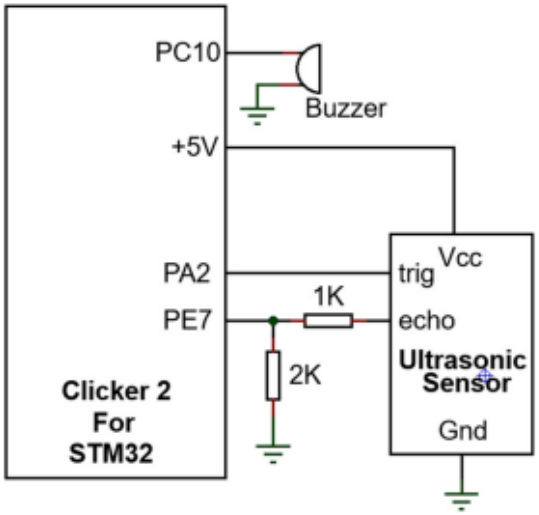

Figura 13: Implementação do divisor de tensão (IBRAHIM,2020)

4.2 Softwares Utilizados

Para implementação do projeto, é necessária a utilização de um ambiente de desenvolvimento integrado (IDE). Foi escolhido o MikroC PRO for ARM, da empresa MikroEletronika, devido a sua grande capacidade de implementação de diversas bibliotecas e interface amigável. O software possui uma licença gratuita, que possui limitações e uma licença paga, onde todos os recursos estão liberados.

Apesar de ser suficiente para tudo no projeto, a versão gratuita não permite que sejam compiladas mais de 250 linhas. Na figura 13 podemos ver como é a interface da IDE.

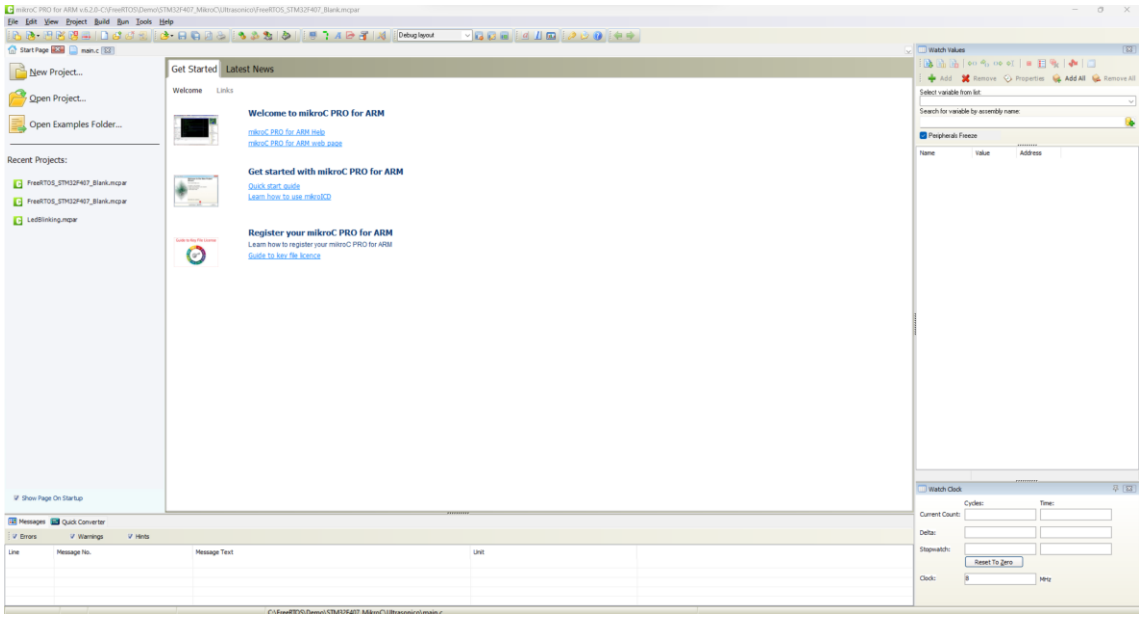

Figura 14: Interface do MikroC PRO for ARM. Arquivo Pessoal

A partir da versão 5.1.0 da IDE, já existe um suporte disponível para sistemas de tempo real, em específico o FreeRTOS.

Além da IDE, foi necessário escolher qual sistema de tempo real utilizar. Para isso, o FreeRTOS foi definido como a melhor escolha, pois além de ser um sistema de licença livre, possui suporte pela IDE.

O FreeRTOS possibilita que seja implementado diversos tipos de funções, em específico o funcionamento com tarefas, relógios, PWM etc. Além disso, possui uma documentação online e de fácil entendimento, o que torna mais simples a sua implementação.

Também foi necessário a utilização de um software que seja capaz de subir os arquivos compilados pelo MikroC PRO for ARM para a placa que contém o microcontrolador. Dessa forma, foi escolhido o *mikroBootloader,* software que também é oferecido pela empresa MikroEletronika. O software possui uma licença gratuita e nenhuma versão paga. Possui um funcionamento simples, com uma interface amigável, como pode ser vista na figura 14.

Figura 15: Interface do mikroBootloader. Arquivo pessoal

4.3 Implementação

Inicialmente foi feita a montagem do *Buggy,* aonde as peças que vieram separadas, foram postas em conjunto. A partir disso, foi instalado o sensor ultrassônico, mostrado na figura 11. Esse sensor é capaz de medir distâncias entre 2cm e 4m, sem nenhum contato e com uma precisão de 3mm. O módulo possui em seu conjunto um receptor e um transmissor ultrassônico, com um circuito interno responsável por administrar o sinal de disparo da medição. O conjunto do HC-SR04 possui quatro pinos: **VCC, *Trigger*, ECHO, GND.**

Para seu devido funcionamento, é necessário que o módulo seja alimentado com um sinal lógico alto de 5V no pino *Trigger* por 10 microssegundos. A partir disso, o módulo gerará um pulso de 40kHz, colocando o pino ECHO em nível lógico alto até que seja detectado o sinal de retorno. Esse tempo será utilizado para calcular a distância entre o ponto medido. É necessário realizar a implementação de um divisor de tensão, como pode ser visto na figura 12, com o pino ECHO do sensor e a GPIO do microcontrolador, pois o sensor ultrassônico gera um sinal de 5V e a porta GPIO possui uma entrada de 3.3V.

A partir da montagem do Buggy com o sensor ultrassônico, dá-se início a parte de implementação, uma vez que o Buggy já possui os demais componentes necessários embarcados em sua placa controladora. Dessa forma, seguimos para a implementação de uma tarefa que é responsável pela realização de algumas rotinas de luzes.

Na figura 15, podemos ver a implementação dos métodos que são responsáveis pelas alterações dos LEDs

```
void LeftSignalFlash(int ratems, int count)
{
    #define LeftSignalLights GPIOC_ODR.B4
    int k;
    GPIO_Config(&GPIOC_BASE, _GPIO_PINMASK_4,_GPIO_CFG_MODE_OUTPUT);

    for(k = 0; k < count; k++)
    {
        LeftSignalLights = 1;
        vTaskDelay(pdMS_TO_TICKS(ratems));
        LeftSignalLights = 0;
        vTaskDelay(pdMS_TO_TICKS(ratems));
    }
}

void RightSignalFlash(int ratems, int count)
{
    #define RightSignalLights GPIOE_ODR.B2
    int k;
    GPIO_Config(&GPIOE_BASE, _GPIO_PINMASK_2,_GPIO_CFG_MODE_OUTPUT);

    for(k = 0; k < count; k++)
    {
        RightSignalLights = 1;
        vTaskDelay(pdMS_TO_TICKS(ratems));
        RightSignalLights = 0;
        vTaskDelay(pdMS_TO_TICKS(ratems));
    }
}

void BrakeLights(int mode)
{
    #define BrakeLight GPIOE_ODR.B1
    GPIO_Config(&GPIOE_BASE,_GPIO_PINMASK_1,_GPIO_CFG_MODE_OUTPUT);
    BrakeLight = mode;
}

void HeadLights(int mode)
{
    #define HeadLight GPIOB_ODR.B6
    GPIO_Config(&GPIOB_BASE, _GPIO_PINMASK_6,_GPIO_CFG_MODE_OUTPUT);
    HeadLight = mode;
```

Figura 16: Implementação das funções de luzes. Arquivo pessoal

Vemos que na função *LeftSignalFlash,* que tem como função fazer com que os LEDs amarelos esquerdo pisquem alternadamente, são definidos dois parâmetros de entrada, sendo eles o *ratems* e o *Count.* O ratems é responsável pela definição em microssegundos de quanto tempo o LED deverá ficar desligado, já o Count é responsável por informar a quantidade de vezes que ação deverá ser repetida. Além disso, é necessário realizar a configuração da GPIO para que seja possível a sua utilização. Define-se uma variável local, chamada de *LeftSignalLights* onde é atribuído o valor do sinal do pino B4 da

GPIO C. Também é feita a configuração da porta, definindo seu modo para OUTPUT. É utilizada a função vTaskDelay que é uma função do FreeRTOS e tem o objetivo de fazer com que a tarefa espere um tempo até que possa ser executada a próxima instrução.

O mesmo procedimento também é executado para função RightSignalFlash, entretanto é definida outra porta GPIO, que sua vez é responsável por fazer com que os LEDs amarelos direitos pisquem alternadamente.

A função *BrakeLights* tem como objetivo ligar os LEDs vermelhos do Buggy, para isso ela possui apenas um parâmetro de entrada, sendo ele o *mode,* sendo ele apenas um valor binário responsável para definir o nível atribuído aos LEDs vermelhos, ligando-os e desligando-os.

A mesma operação é realizada na função HeadLights, entretanto possui como objetivo acionar os LEDs brancos do Buggy.

```
void task1(void *pvParameters)
{
    while (1)
    {
        LeftSignalFlash(100, 5);
        RightSignalFlash(100, 5);
        BrakeLights(1);
        vTaskDelay(pdMS_TO_TICKS(1000));
        BrakeLights(0);
        HeadLights(1);
        vTaskDelay(pdMS_TO_TICKS(1000));
        HeadLights(0);
    }
}
```

Figura 17: Declaração da tarefa 1. Arquivo Pessoal

Na figura 16 é possível ver a declaração da tarefa que será responsável por executar as rotinas estabelecidas anteriormente, definindo seus parâmetros e ordem de execução, no cenário acima, o sistema estará piscando os LEDS amarelos esquerdos 5 vezes com um intervalo de 100ms, em seguida executará a mesma operação para os LEDs amarelos direitos, passará o valor alto para os LEDs vermelhos, aguardará 1000ms e passará o valor baixo para os LEDs

vermelhos, por fim colocará um sinal alto nos LEDS brancos, aguardará 1000ms e em seguida o valor baixo para os LEDs brancos.

```
void main()
{
    // Create task 1.
    xTaskCreate(
        (TaskFunction_t)task1,
        "Luzes",
        configMINIMAL_STACK_SIZE,
        NULL,
        10,
        NULL
    );

    // Start the RTOS scheduler.
    vTaskStartScheduler();

    // Will never reach here.
    while (1);
}
```

Figura 18: Criando a tarefa 1 no FreeRTOS. *Arquivo* Pessoal

Na figura 17, é evidenciado a declaração da tarefa 1 no FreeRTOS, demos o nome de “Luzes”, com uma prioridade 10. Em seguida, iniciamos o escalonador do FreeRTOS.

```
PWMA = PWM_TIM11_Init(1000);    // For pin PB9
PWMB = PWM_TIM10_Init(1000);    // For pin PB8
PWMC = PWM_TIM9_Init(1000);    // For pin PE5
PWMD = PWM_TIM3_Init(1000);    // For pin PB0

PWM_TIM11_Start(_PWM_CHANNEL1, &_GPIO_MODULE_TIM11_CH1_PB9);
PWM_TIM10_Start(_PWM_CHANNEL1, &_GPIO_MODULE_TIM10_CH1_PB8);
PWM_TIM9_Start(_PWM_CHANNEL1, &_GPIO_MODULE_TIM9_CH1_PE5);
PWM_TIM3_Start(_PWM_CHANNEL3, &_GPIO_MODULE_TIM3_CH3_PB0);
```

Figura 19: Inicialização do PWMs. Arquivo Pessoal

Na figura 18, pode ser visto a inicialização dos PWMs que serão utilizados no projeto, são declaradas 4 variáveis globais, sendo elas: PWMA, PWMB, PWMC, PWMD. Para elas são definidos 4 timers diferentes de PWM, em seguida

realizamos a inicialização de cada um desses PWMs, configurando os canais e as portas de propósito geral que serão utilizadas.

```
void MoveForward(int n, int mode)
{
    int secs;
    secs = n * 50;
    PWM_TIM11_Set_Duty(0,_PWM_NON_INVERTED, _PWM_CHANNEL1);
    PWM_TIM9_Set_Duty(0, _PWM_NON_INVERTED, _PWM_CHANNEL1);
    PWM_TIM3_Set_Duty(PWMD/mode,_PWM_NON_INVERTED,_PWM_CHANNEL3);
    PWM_TIM10_Set_Duty(PWMB/mode,_PWM_NON_INVERTED, _PWM_CHANNEL1);
    vTaskDelay(pdMS_TO_TICKS(secs));
    Stop();
}

 void MoveBackwards(int n, int mode)
{
    int secs;
    secs = n * 50;
    PWM_TIM3_Set_Duty(0,_PWM_NON_INVERTED, _PWM_CHANNEL3);
    PWM_TIM10_Set_Duty(0, _PWM_NON_INVERTED, _PWM_CHANNEL1);
    PWM_TIM11_Set_Duty(PWMA/mode,_PWM_NON_INVERTED,_PWM_CHANNEL1);
    PWM_TIM9_Set_Duty(PWMC/mode,_PWM_NON_INVERTED, _PWM_CHANNEL1);
    vTaskDelay(pdMS_TO_TICKS(secs));
    Stop();
}

void TurnLeft()
{
    PWM_TIM11_Set_Duty(0,_PWM_NON_INVERTED, _PWM_CHANNEL1);
    PWM_TIM10_Set_Duty(0, _PWM_NON_INVERTED, _PWM_CHANNEL1);
    PWM_TIM9_Set_Duty(0,_PWM_NON_INVERTED,_PWM_CHANNEL1);
    PWM_TIM3_Set_Duty(PWMD,_PWM_NON_INVERTED, _PWM_CHANNEL3);
    vTaskDelay(pdMS_TO_TICKS(25));
    Stop();
}

 void TurnRight()
{
    PWM_TIM11_Set_Duty(0,_PWM_NON_INVERTED, _PWM_CHANNEL1);
    PWM_TIM10_Set_Duty(PWMB, _PWM_NON_INVERTED, _PWM_CHANNEL1);
    PWM_TIM9_Set_Duty(0,_PWM_NON_INVERTED,_PWM_CHANNEL1);
    PWM_TIM3_Set_Duty(0,_PWM_NON_INVERTED, _PWM_CHANNEL3);
    vTaskDelay(pdMS_TO_TICKS(25));
    Stop();
```

Figura 20: Funcionamento dos motores do Buggy. Arquivo Pessoal

Na Figura 19, é exemplificado a implementação das funções de locomoção do Buggy. Como o veículo não possui eixos, para realizar as curvas é necessário ativar os motores do lado oposto ao que se deseja virar. Para isso foram utilizados 4 PWMs, responsáveis por controlar cada motor do Buggy, com eles é possível definir o tempo de trabalho e a velocidade da execução.

Na função MoveForward, que possui como objetivo fazer com que o Buggy se mova para frente, são passados dois parâmetros, sendo eles *n* e *mode* onde *n* é responsável por definir o tempo de execução em segundos e mode é responsável por definir o tempo de trabalho dos PWMs, que serão inversamente proporcionais ao valor informado, tornando o Buggy mais rápido ou mais lento.

A função MoveBackwards possui as mesmas entradas, porém irá inverter o sentido do movimento, fazendo com que o robô se mova para trás.

Já a função MoveRight tem como objetivo fazer com que o Buggy vire para o lado direito, para isso ela não possui parâmetros de entrada, ativará apenas o motor dianteiro esquerdo, fazendo com que o veículo faça um giro de 90 graus para o lado direito. Ressalta-se que os testes foram realizados em uma superfície plana, com boa aderência, seca e sem irregularidades.

A função MoveLeft possui o mesmo princípio da MoveRight, entretanto, ativará o motor dianteiro direito, fazendo com que o veículo faça uma curva de 90 graus para esquerda. As mesmas irregularidades foram identificadas para esse movimento.

```
void task1(void *pvParameters)
{
    while (1)
    {
        MoveForward(1,4);
        vTaskDelay(pdMS_TO_TICKS(100));
        TurnLeft();
        vTaskDelay(pdMS_TO_TICKS(100));
        MoveBackwards(1,4);
        vTaskDelay(pdMS_TO_TICKS(100));
        TurnRight();
        vTaskDelay(pdMS_TO_TICKS(100));
        MoveForward(1,4);
        vTaskDelay(pdMS_TO_TICKS(100));
    }
}
```

Figura 21: Declaração da tarefa do movimento. Arquivo Pessoal

Na figura 20 vemos a declaração da tarefa que será responsável pela movimentação do veículo, no procedimento foi definido que o veículo fara um movimento para frente por 1 segundo, uma curva de 90 graus para esquerda, um movimento para trás por 1 segundo, uma curva de 90 graus para direita e um movimento para frente por 1 segundo. Entre cada movimento, serão esperados 1000ms.

A partir disso, é possível a execução de duas tarefas no escalonador do FreeRTOS, para isso é feita a declaração das duas, como pode ser feito na figura 21.

```
    xTaskCreate(
        (TaskFunction_t)task1,
        "Motores",
        configMINIMAL_STACK_SIZE,
        NULL,
        10,
        NULL
    );
    xTaskCreate(
        (TaskFunction_t)task2,
        "Luzes",
        configMINIMAL_STACK_SIZE,
        NULL,
        10,
        NULL
    );

    // Start the RTOS scheduler.
    vTaskStartScheduler();

    // Will never reach here.
    while (1);
}
```

Figura 22: criação das tarefas. Arquivo Pessoal

A tarefa para a execução dos motores é declarada com o nome de Motores e possui uma prioridade igual à da tarefa das luzes. Com isso, o escalonador será responsável por definir qual tarefa será executada em determinado momento.

```
{
  #define trig GPIOA_ODR.B2
  #define echo GPIOE_IDR.B7

  float Elapsed;
  unsigned int Present, IntDistance;
  char Txt[14];

  UART3_Init_Advanced(9600,_UART_8_BIT_DATA,_UART_NOPARITY,_UART_ONE_STOPBIT,
                      & _GPIO_MODULE_USART3_PD89);
  GPIO_Config(&GPIOA_BASE, _GPIO_PINMASK_2, _GPIO_CFG_MODE_OUTPUT);
  GPIO_Config(&GPIOE_BASE, _GPIO_PINMASK_7, _GPIO_CFG_MODE_INPUT);
  trig = 0;

  RCC_APB1ENR.TIM2EN = 1; //Ativa o clock para o Timer 2
  TIM2_CR1.CEN = 0; //Desativa o contador do Timer 2
  TIM2_PSC = 1000; //Define o Prescaler em 1000
  TIM2_ARR = 65535; //Define o registrador de auto-reload

  while(1)
  {
        BrakeLights(1);
        TIM2_CNT = 0; //Define o contador em 0
        trig = 1; //Manda o pulso trig
        Delay_us(10); //Espera 10 micro segundos
        //vTaskPrioritySet(NULL , 11); // Aumenta a prioridade
        HeadLights(0);
        while(echo == 0); // Espera pela borda subindo
        TIM2_CR1.CEN = 1; //Inicia o contador
        while(echo == 1); //Aguarda pela borda descendo
        TIM2_CR1.CEN = 0; //Para o contador
        HeadLights(1);
       // vTaskPrioritySet(NULL , 10); // Volta a prioridade para o normal
        Present = TIM2_CNT; //Pega o valor que esta no contador
        Elapsed = Present * 11.9047; //Calcula o tempo do pulso
        Elapsed = 0.017 * Elapsed; //Obtem a distancia em CM
        IntDistance = (int)Elapsed; //Distance como inteiro
        RightSignalFlash(IntDistance / 10,2);
        BrakeLights(0);
        HeadLights(0);
        vTaskDelay(pdMS_TO_TICKS(50)); //Espera 50ms
```

Figura 23: Tarefa do sensor ultrassônico. Arquivo pessoal

Na figura 22, podemos ver a definição a tarefa utilizada para o funcionamento do sensor ultrassônico na aplicação. Inicialmente é necessário definir quais portas serão utilizadas para a entrada e a saída dos dados. A partir disso, será necessário utilizar alguns clocks e timers para a operação correta do sensor.

Nessa tarefa, inicialmente colocaremos o valor 0 no Contador do timer 2, isso é necessário para garantir que a contagem do timer estará em 0, em seguida, será necessário enviar o pulso **TRIG,** para isso, iremos colocar o valor 1 na variável que definimos em relação a porta de saída. Com o sinal emitido, é necessário esperar o retorno do **ECHO,** quando a borta do pulso for 0, ou seja, subindo, iniciaremos o Contador e o manteremos ativado até a borda estar com o valor 1.

Com os valores dos contadores e timers obtidos, é possível realizar o cálculo da distância medida, pra isso, é utilizada a seguinte formula já detalhada anteriormente nesse documento: Elapsed = ValorContador * 11.9047; Com isso é possível descobrir o tempo em que o pulso demorou para retornar. Com esses

dados é possível utilizar a fórmula: Elapsed = 0.017 * Elapsed; Assim iremos obter a distância calculada em centímetros.

Após essas operações, é necessário verificar que a distância mensurada seja uma variável do tipo inteiro, para isso é necessário realizar uma conversão. Nessa conversão, existe uma pequena perda de precisão, uma vez que a distância terá apenas o seu valor inteiro e perderá suas casas decimais. Isso é necessário, pois os parâmetros dos métodos esperam a entrada de um tipo inteiro para o cálculo do tempo.

No procedimento realizado, foram encontrados alguns problemas, uma vez em que o sistema só consegue realizar a operação uma única vez, ou seja, apesar da tarefa estar definida do FreeRTOS, ela executa corretamente apenas na primeira execução do programa, não conseguindo realizar uma nova medição para os dados futuros.

5. CONCLUSÃO

Considerando os objetivos propostos no trabalho, pode-se concluir que eles foram parcialmente concluídos. Foi possível realizar a construção de um dispositivo controlado por um microprocessador ARM capaz de executar um Sistema de tempo real e seu escalonador.

Foi possível realizar a implementação de um sensor ultrassônico que fosse capaz de identificar a distância de um objeto a sua frente. Entretanto, houve alguns contratempos na implementação. O código utilizado para a implementação do sensor funcionou corretamente, porém, ele é capaz de realizar a leitura apenas uma única vez, fazendo necessário que para cada releitura o robô seja reiniciado.

A dificuldade do sensor ultrassônico poderia ser evitada se utilizado um fornecido pela empresa desenvolvedora do kit *Buggy*, a MikroEletronika. Entretanto, devido a custos, processos de importação e por se tratar de um projeto experimental, foi mais viável a escolha de um sensor de outra marca que não possuía um suporte oficial da empresa. Foi necessário realizar algumas adaptações no projeto para que o sensor pudesse ser encaixado na parte frontal do veículo, para isso foi utilizado uma protoboard com fita dupla face na sua parte inferior.

Para a realização dos testes com o robô, foi utilizado uma superfície plana, seca e sem nenhuma irregularidade, contudo, apesar das mesmas condições de teste, o movimento de curva do robô não possuía uma precisão alta, devido ao fato dele não possuir eixos para realizá-las. Devido a essa característica, foi necessário otimizar o tempo de trabalho dos PWMs para garantir um resultado confiável.

O teste de tempo real também foi realizado na prática. Como as tarefas que estavam executando eram simples, o escalonador conseguia executá-las sem nenhum problema, uma vez que a prioridade definida era sempre a mesma, tanto as luzes quanto os motores eram acionados quando esperados.

5.1 Trabalhos futuros

Devido à complexidade do desenvolvimento de um sistema descrito nesse trabalho, alguns pontos podem ser melhorados e aprimorados para que o

funcionamento seja mais otimizado. É possível citar algumas melhorias técnicas que podem aprimorar a implementação do *Buggy.*

5.2 Melhorias técnicas

Aplicar um sensor ultrassônico fabricado pela própria MikroEletronika seria fundamental para esse projeto, pois apesar de ser possível implementar um sensor externo, o sensor fornecido pela empresa fabricante já possui os protocolos de comunicação com o BUS do carrinho robô, tornando mais fácil sua implementação e garantia de funcionamento adequado.

Outro ponto interessante seria a implementação de novos sensores, tanto sensores fotossensíveis, que podem ser utilizados para o funcionamento dos LEDs do robô, quanto de módulos GPS, para que o robô seja capaz de saber precisamente sua posição no espaço. Esses módulos também são disponibilizados pela MikroEletronika em seu site e possuem fácil adaptação com o kit de desenvolvimento *Buggy.*

Por fim, uma outra melhoria técnica que pode ser implementada, porém com um grau de complexidade maior é a instalação de um eixo no carrinho, o que tornaria muito mais preciso a implementação de curvas, uma vez que utilizando um servo motor seria capaz de girar apenas o necessário para cada curvatura, não sendo necessário ajustar precisamente o tempo de trabalho de cada PWM de forma individual.

REFERÊNCIAS

BRÄUNL, Thomas. *Embedded Robotics: From Mobile Robots to Autonomous Vehicles with Raspberry Pi and Arduino*. 4. ed. Perth, WA, Australia: School of Engineering, The University of Western Australia, 123.

Geeks for Geeks. *Real Time Operating System (RTOS).* 2022. Disponível em: < https://www.geeksforgeeks.org/real-time-operating-system-rtos/ > Acesso em 11 de outubro de 2022

GIL, Antônio Carlos. *Como Elaborar Projetos de Pesquisa***.** 6. ed. São Paulo: Editora Atlas Ltda., 2017.

IBRAHIM, Dogan. *ARM-Based Microcontroller Multitasking Projects Using the FreeRTOS Multitasking Kernel*. Newnes, Elsevier, 2020.

OMS. *Salvar VIDAS – Pacote de medidas técnicas para a segurança no trânsito*. Brasília, DF: OPS; 2018. Licença: CC BY-NC-SA 3.0 IGO.

OPAS. *Segurança no Trânsito*. 2018. Disponível em: < https://www.paho.org/pt/topicos/seguranca-no-transito> Acesso em 11 de outubro de 2022

Red Hat. *What is an ARM processor?* 2022. Disponível em: <https://www.redhat.com/en/topics/linux/what-is-arm-processor#what-makes-the-arm-architecture-valuable > Acesso em 11 de outubro de 2022

SAE. *SAE Levels of Driving Automation™ Refined for Clarity and International Audience***.** 2021. Disponível em < https://www.sae.org/blog/sae-j3016-update#:~:text=With%20a%20taxonomy%20for%20SAE's,and%20their%20operation%20on%20roadways > Acesso em 11 de outubro de 2022

WAZLAWICK, R. S. *Metodologia da Pesquisa para Ciência da Computação*. 2ª. ed. [S.l.]: Campus, 2014.

WEI, M.C. D. *Método de desvio de obstáculo aplicado a veículo autônomo*. Mestrado em Engenharia de transportes, Escola politécnica da Universidade de São Paulo, São Paulo. 2015

Wind River. *What is a Real-Time Operating System (RTOS)?* 2022. Disponível em: < https://www.windriver.com/solutions/learning/rtos > Acesso em 11 de outubro de 2022

PUC GOIÁS

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
GABINETE DO REITOR

Av. Universitária, 1069 • Setor Universitário
Caixa Postal 86 • CEP 74605-010
Goiânia • Goiás • Brasil
Fone: (62) 3946.1000
www.pucgoias.edu.br • reitoria@pucgoias.edu.br

RESOLUÇÃO nº 038/2020 – CEPE

ANEXO I

APÊNDICE ao TCC

Termo de autorização de publicação de produção acadêmica

O(A) estudante Matheus Dias Ferlin Moura do Curso de Engenharia da Computação ,matrícula 201800330050-1 ,telefone: 62982291167 e-mail mdfm9@hotmail.com, na qualidade de titular dos direitos autorais, em consonância com a Lei nº 9.610/98 (Lei dos Direitos do Autor), autoriza a Pontifícia Universidade Católica de Goiás (PUC Goiás) a disponibilizar o Trabalho de Conclusão de Curso intitulado Desenvolvimento de carro robô autônomo com sistema de tempo-real, gratuitamente, sem ressarcimento dos direitos autorais, por 5 (cinco) anos, conforme permissões do documento, em meio eletrônico, na rede mundial de computadores, no formato especificado (Texto(PDF); Imagem (GIF ou JPEG); Som (WAVE, MPEG, AIFF, SND); Vídeo (MPEG, MWV, AVI, QT); outros, específicos da área; para fins de leitura e/ou impressão pela internet, a título de divulgação da produção científica gerada nos cursos de graduação da PUC Goiás.

Goiânia, 26 de junho de 2023.

Assinatura do autor: Matheus D F Moura

Nome completo do autor: Matheus Dias Ferlin Moura

Assinatura do professor–orientador: Sibelius Lellis Vieira

Nome completo do professor-orientador: Sibelius Lellis Vieira