PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
ENGENHARIA DE COMPUTAÇÃO

THIAGO WARTCHOW

ALIMENTADOR AUTOMÁTICO PARA ANIMAIS DE PEQUENO PORTE

Porto Alegre
2024

THIAGO WARTCHOW

ALIMENTADOR AUTOMÁTICO PARA ANIMAIS DE PEQUENO PORTE

Trabalho de conclusão de curso de graduação apresentado na Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul, como requisito parcial para obtenção do grau de Engenheiro de Computação.

Orientador: Marlon Leandro Moraes

Porto Alegre

2024

THIAGO WARTCHOW

ALIMENTADOR AUTOMÁTICO PARA ANIMAIS DE PEQUENO PORTE

Trabalho de conclusão de curso de graduação apresentado na Escola politécnica da Pontifícia Universidade Católica do Rio Grande do Sul, como requisito parcial para obtenção do grau de Engenheiro de Computação.

Aprovada em _____ de _____ de _____.

BANCA EXAMINADORA:

Prof. Me. Júlio César Marques de Lima

_____

ORIENTADOR:

Prof. Me. Marlon Leandro Moraes

_____

Dedico este trabalho a minha
família.

AGRADECIMENTOS

Agradeço primeiramente ao meu orientador, pela recomendação do tema do projeto em questão, o ensino prévio em disciplinas ao longo do curso e o auxílio na execução do projeto ao longo do semestre.

Sou grato a meus pais que me deram a oportunidade de estudar em uma faculdade de renome e ao apoio durante o curso e a esta jornada que está se aproxima de seu fim.

Também agradeço aos professores com quem convivi e me ensinaram tanto nos últimos 5 anos.

“O insucesso é apenas uma oportunidade para recomeçar com mais inteligência.”

Henry Ford

RESUMO

Este projeto visa automatizar o processo de alimentação de animais de pequeno porte para proprietários de animais que não tenham disponibilidade de se manter em suas residências durante todo o dia. O alimentador proporciona a opção de alimentar o *pet* diariamente com a quantidade especificada pelo proprietário e agendar horários para ativação do alimentador, juntamente com a quantidade que será distribuída para o *pet*. Este agendamento é feito através de uma página *web* de fácil acesso e intuitiva.

O projeto faz uso de dois principais agentes: um microcontrolador ESP32, que gerencia a criação da página *web* e armazena os horários de alimentação determinados pelo usuário, e um motor de passo que, como atuador, fará a distribuição do alimento de acordo com as determinações do usuário. Neste projeto, o motor atuador é controlado pelo microcontrolador através de um *firmware* desenvolvido em linguagem C++.

Palavras-chave: Alimentador automático para *pets*. Controle remoto. Microcontrolador.

ABSTRACT

This project aims to automate the feeding process for small pets for pet owners who are not able to stay at home throughout the day. The feeder provides the option to feed the pet automatically with the specified amount set by the owner and to schedule feeding times along with the amount to be dispensed for the pet. This scheduling is done through an easy-to-access and intuitive web page.

The project utilizes two main components: an ESP32 microcontroller, which manages the creation of the web page and stores the feeding times set by the user, and a stepper motor that, as an actuator, will dispense the food according to the user's specifications. In this project, the actuator motor is controlled by the microcontroller through firmware developed in C++ language.

Keywords: Pets. Automatic feeder. Remote control. Microcontroller.

LISTA DE SIGLAS

ESP32 – Microcontrolador Espressif Systems

IoT – *Internet of Things*

GPIO - *General Purpose Input/Output*

I/O – *Input/Output*

SUMÁRIO

1 INTRODUÇÃO

De acordo com a Euromonitor Internacional, empresa mundial de inteligência de negócios e análise de mercado, no ano de 2021 a população de animais de companhia no Brasil, como aves, cães e gatos, era de 149,6 milhões, colocando o Brasil em terceiro lugar entre os países com maior número de animais domésticos, perdendo apenas para China e os Estados Unidos. Essa população de animais de estimação vem crescendo a cada ano como mostra a Quadro 1.

Ano	Cães	Gatos	Aves	Peixes	Rep. Peq. Mam.	Total
2018	54,2	23,9	39,8	19,1	2,3	139,3
2019	55,1	24,7	40	19,4	2,4	141,6
2020	55,9	25,6	40,4	19,9	2,51	144,31
2021	58,1	27,1	41	20,8	2,53	149,53

Quadro 1 - Crescimento de pets no Brasil de 2018 - 2021 (IPB, Instituto Pet Brasil).

Com o crescente aumento de animais, veio junto a mudança do papel deles dentro dos lares. Em uma pesquisa realizada no ano de 2019 pela Organização Mundial de Proteção aos Animais, o Brasil se destacou por ser o país que têm mais cães, e 94% das pessoas consideram seus *pets* como membros da família.

Levando isso em conta, a necessidade de uma alimentação de qualidade e regrada se torna necessária para manter a saúde dos *pets*. Boa parte dos animais de estimação acabam ingerindo muito mais calorias do que deveriam, não respeitando os limites da saciedade. Por isso, é importante controlar a quantidade das porções de comida oferecidos a eles, para prevenir o excesso de peso e problemas de saúde relacionados.

Com isso em mente, o alimentador automático é uma possível solução para oferecer os meios necessários para controlar a quantidade de alimento que o animal irá consumir, sem que o proprietário do animal tenha que estar presente a todo momento.

Nota-se que este projeto consiste em apenas um protótipo que serve para demonstrar que executar uma tarefa diária não precisa se tornar um incômodo. Algumas funcionalidades essenciais em um produto comercializável não foram implementadas, portanto, o projeto tem como finalidade estudo e a aplicação de conhecimentos adquiridos durante o curso de engenharia da computação.

1.2 Objetivo

O objetivo do projeto é desenvolver um alimentador automático que distribua ração para o animal e possa ser controlado remotamente através de um *smartphone* que esteja conectado a mesma rede *Wi-Fi* do microcontrolador ESP32.

3 MERCADO PET

3.1 Futuro do mercado *pet*

O segmento *pet* possui um público fiel e em constante crescimento. De acordo com o SEBRAE existem cerca de 285 mil empresas voltadas para os *pets* no Brasil, sendo o terceiro maior mercado *pet* do mundo.

Ao contrário da queda que ocorreu nos outros setores durante a pandemia, as empresas de cuidados com animais domésticos estiveram em constante crescimento e ampliação de seu mercado, isso porque muitas pessoas adotaram ou compraram *pets* para ter uma companhia durante o período de isolamento.

Fora a aquisição dos *pets* com objetivo de companheirismo, os animais, principalmente cachorros, são treinados em diversas áreas para servir aos humanos, como cães farejadores, que trabalham com a polícia por exemplo, os *service dogs* (normalmente designados a pessoas com necessidades especiais, como os deficientes visuais), e até mesmo para proteção dos donos, seja em sua residência ou até mesmo na rua.

Além disso, havia uma maior convivência com os *pets* durante a quarentena, o que aumentou ainda mais o afeto pelos animais de estimação, tornando os cuidados com os *pets* cada vez mais almejados pelos donos, fazendo com que os serviços e espaços para os *pets* tenham que se adequar a essa necessidade.

Com isso surgiram novas possibilidades de negócio voltadas a esse cuidado mais especializado, como: refeição natural, creches, lavanderias especializadas, SPAs, festas para *pets*, serviço funeral e plano de saúde, *dog walker* (passeador de cães), *pet sitter* (babá de *pet*) e versões *pet friendly* de comidas.

3.2 Dispositivos de alimentação automatizada para *pets*

3.2.1 Benefícios da utilização

Com o avanço da tecnologia, a vida dos animais de estimação e de seus tutores tem sido cada vez mais facilitada por dispositivos, como os sistemas de alimentação automatizada para *pets*, também conhecidos como “comedouros automáticos”. Estes sistemas são projetados para fornecer comida aos animais de estimação de forma programada e precisa, sendo concebidos em várias formas e tamanhos, adequando-se a diferentes necessidades e tipos de animais, desde pequenos gatos até grandes cães.

Uma das maiores vantagens dos dispositivos de alimentação automatizada é a conveniência que eles oferecem aos tutores dos animais de estimação, que muitas vezes não conseguem estar em casa em horários específicos para alimentar seus animais.

O uso do alimentador automático é particularmente importante para animais com necessidades dietéticas especiais, como os que têm problemas de sobrepeso ou que precisam de refeições regulares para controlar condições médicas, além de também ajudarem animais de estimação com ansiedade de separação de seus tutores.

É importante ressaltar que todo o animal está sujeito a sofrer de sobrepeso ou obesidade caso a quantidade de calorias consumidas esteja acima do recomendado. Isso pode resultar em uma expectativa de vida reduzida comparado com os animais que estão dentro do peso ideal.

De acordo com o AVMA (*American Veterinary Medical Association*), uma pesquisa com dados fornecidos pelo *Banfield Pet Hospital* descobriu que a expectativa de vida de cães com sobrepeso era, em média, até 2 anos e meio mais curta do que a expectativa de vida de cães com peso corporal saudável.

O estudo, realizado na Inglaterra pela Universidade de Liverpool e pelo *Waltham Centre for Pet Nutrition* da *Mars Petcare*, revelou que cães com sobrepeso, em geral, têm mais chances de ter uma vida mais curta do que aqueles com peso corporal ideal. O estudo, intitulado *"Associação entre expectativa de vida e condição corporal em cães castrados de clientes"*, foi conduzido retrospectivamente ao longo de duas décadas e publicado online em 11 de dezembro no *Journal of Veterinary Internal Medicine*.

Outra característica desses dispositivos é a capacidade de monitorar e controlar a alimentação à distância, usando aplicativos móveis conectados via *Wi-Fi* ou *Bluetooth*. Isso permite que os donos de animais de estimação ajustem a

programação de alimentação de acordo com as necessidades do animal, mesmo quando não estão em casa. Alguns modelos até permitem a liberação de alimentos por comando de voz, tornando o processo ainda mais interativo.

A segurança é uma prioridade ao se escolher um dispositivo de alimentação automatizada, a maioria dos produtos é projetada para evitar travamentos ou derramamentos, garantindo que a comida esteja sempre acessível para o *pet*. Além disso, muitos modelos têm baterias de reserva ou *backup* em caso de falhas de energia, garantindo que o animal não fique sem comida, mesmo durante quedas de energia. Outra inovação notável é a inclusão de câmeras de monitoramento em alguns dispositivos. Isso permite que os donos de animais de estimação vejam seus *pets* enquanto se alimentam, verifiquem se eles estão comendo bem e até mesmo interajam com eles por meio de alto-falantes.

3.2.2 Estrutura de um alimentador automático

Normalmente um alimentador automático é composto por seis partes principais: um armazenamento para o alimento, um meio de transporte para o alimento armazenado até um recipiente, uma interface para que o dono do animal possa controlar o alimentador, um recipiente para o alimento que será distribuído, uma fonte de energia e um motor para mover as hélices que transportam o alimento. Podemos observar na Tabela 1 as partes mencionadas.

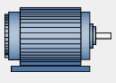
ARMAZENAMENTO	TRANSPORTE	INTERFACE USUÁRIO	RECIPIENTE	FONTE DE ENERGIA	MOTORIZAÇÃO
 Recipiente de Plástico	 Rosca Helicoidal  Válvula rotativa	 Smartphone  Temporizador	 Pote de plástico	 Bateria  Carregador	 Motor de passo  Motor elétrico

Tabela 1 - Partes de um alimentador automático. Fonte: autor

3 PROTÓTIPO DO ALIMENTADOR

3.1 Projeto conceitual

A ideia do projeto foi de criar um alimentador que permita o usuário alimentar seu *pet* de forma manual ou agendada.

O alimentador se comporta da seguinte forma:

- Através da interface de usuário, o tutor do *pet* pode escolher a quantidade de alimento distribuída ao animal de forma manual ou determinar um ou mais horários em que o alimentador será acionado automaticamente.
- O microcontrolador ESP32, escolhido para esse projeto por ter suporte a *Wi-Fi*, receberá as informações fornecidas pelo usuário através da página *web*, ativando o alimentador instantaneamente com a quantidade de alimento escolhida pelo usuário caso a opção de alimentação manual seja escolhida. Caso a opção de agendamento seja usada, o ESP32 armazena o horário que o usuário determinou para alimentação e através de uma comparação com o horário atual e os horários salvos ele acionará o alimentador nos horários agendados.
- O motor de passo, quando acionado pelo ESP32, fará com que a hélice helicoidal gire no sentido horário, despejando o alimento no recipiente utilizado para alimentar o *pet*.

  Na Figura 1 pode-se observar o fluxograma do comportamento esperado conforme itens detalhados anteriormente neste tópico.

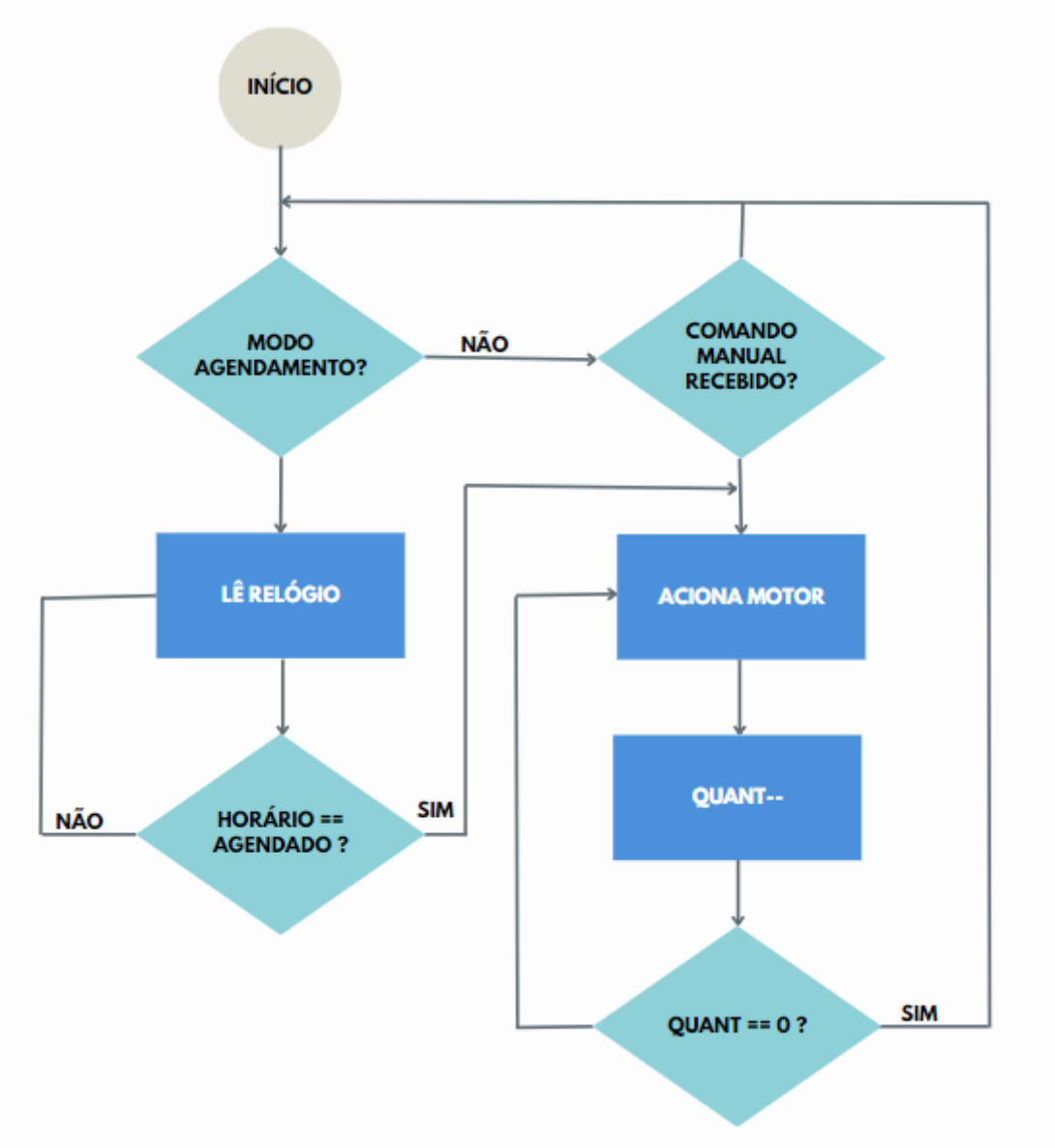


Figura 1 - Fluxograma do alimentador automático. Fonte: autor

3.2 Estrutura

O modelo 3D utilizado neste projeto foi obtido a partir do site *Smart Solutions for Home*. Na Figura 2 pode-se observar a estrutura do modelo 3D já montada e algumas possíveis variantes.

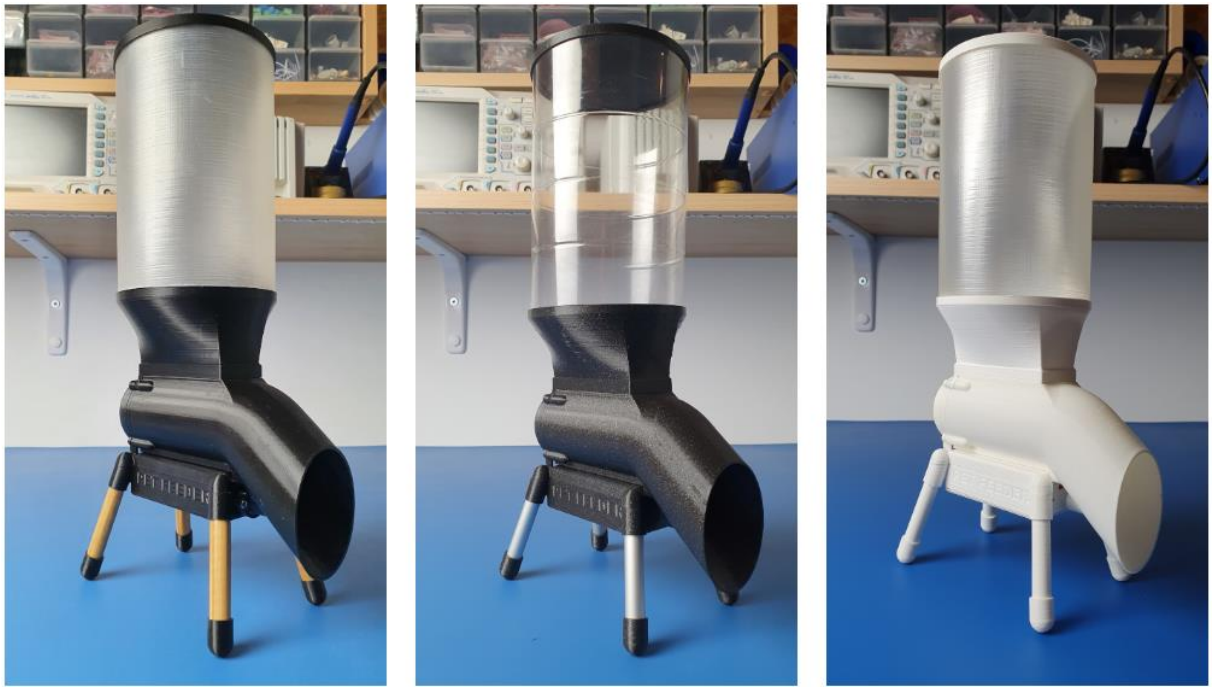


Figura 2 - Alimentador montado. Fonte: site *Smart Solutions for Home*

A Figura 3 apresenta as partes do modelo 3D após a impressão.

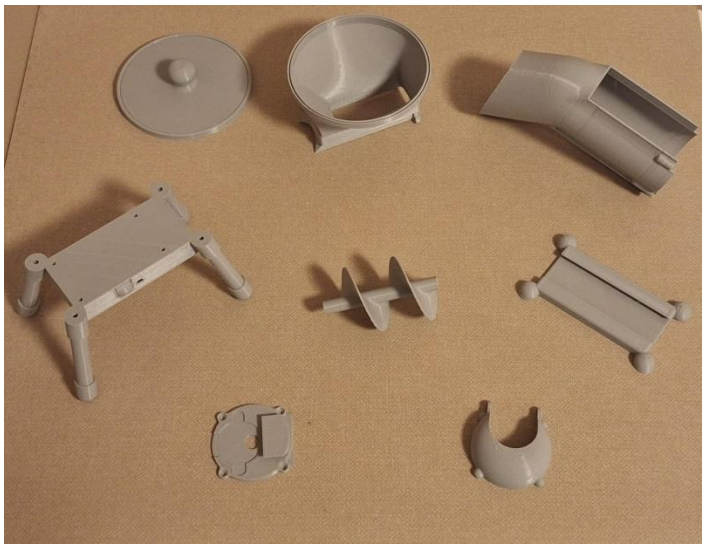


Figura 3 - Alimentador desmontado. Fonte: autor

3.3 Hardware

3.3.1 Esp32

O ESP32 é um microcontrolador desenvolvido pela Espressif Systems. E lançado em 2016. Ele é a evolução do ESP8266 e oferece conectividade *Wi-Fi* e

Bluetooth integradas, bem como maior poder de processamento e mais recursos, tornando-o amplamente utilizado em projetos de IoT e eletrônicos. Desde seu lançamento, o ESP32 ganhou popularidade na comunidade de desenvolvedores devido à sua versatilidade e recursos avançados.

O Quadro 2 faz uma comparação entre seu antecessor, bem como suas especificações técnicas.

	ESP8266	**ESP32**
Corrente	197mA	220mA
Núcleo	1	2
Arquitetura	32 bits	32 bits
Clock	80 – 160 MHz	160-240 MHz
Bluetooth	Não	Clássico e BLE (Bluetooth Low Energy)
WiFi	Sim	Sim
RAM	160KB	520KB
FLASH	16Mb	16Mb
GPIO	13	34
DAC	0	2
ADC	1	18
Interfaces	SPI, I2C, UART e I2S	SPI, I2C, UART, I2S e CAN

Quadro 2 - Comparação entre ESP32 e ESP8266. Fonte: Lobo da informática.

Já a Figura 4 mostra o aspecto físico do kit de desenvolvimento do ESP32 e o que significa cada piano de I/O.

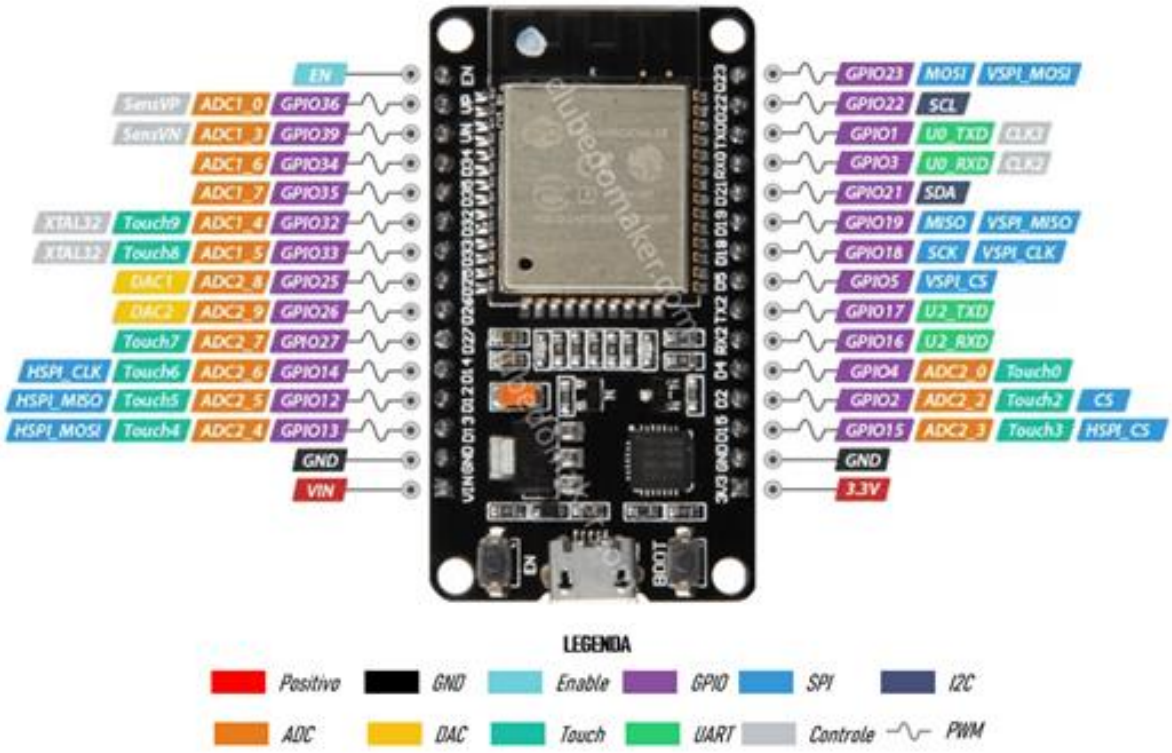


Figura 4 - ESP32 Pinout: Detalhes e Conexões. Fonte: site Clube do Maker.

3.3.2 Motor de passo 28BYJ-48 5V e driver ULN2003

Os motores de passo são utilizados em aplicações industriais e comerciais cotidianas em função de seu baixo custo, da alta confiabilidade, que os permitem ser utilizados em quase qualquer tipo de ambiente.

O 28BYJ-48 é um motor de passo unipolar que opera com tensão de 5V, seu eixo pode ser posicionado com precisão (um passo de cada vez).

O motor de passo 28BYJ-48 possui uma caixa de redução de 1/64 o que significa que é possível dar uma volta completa com 2048 passos, ou seja, aproximadamente 0,18° por passo, com um torque elevado para as dimensões do motor.

Ele possui um total de quatro bobinas. Uma extremidade das bobinas está conectada a 5V, que corresponde ao fio vermelho do motor. As outras extremidades das bobinas correspondem aos fios nas cores azul, rosa, amarelo e laranja. Energizar as bobinas em uma sequência lógica faz o motor mover um passo em uma direção ou na outra. As Figura 5 e 6 foram colocadas para facilitar a compreensão da explicação, e mostram o diagrama de conexões e o aspecto físico deste motor.

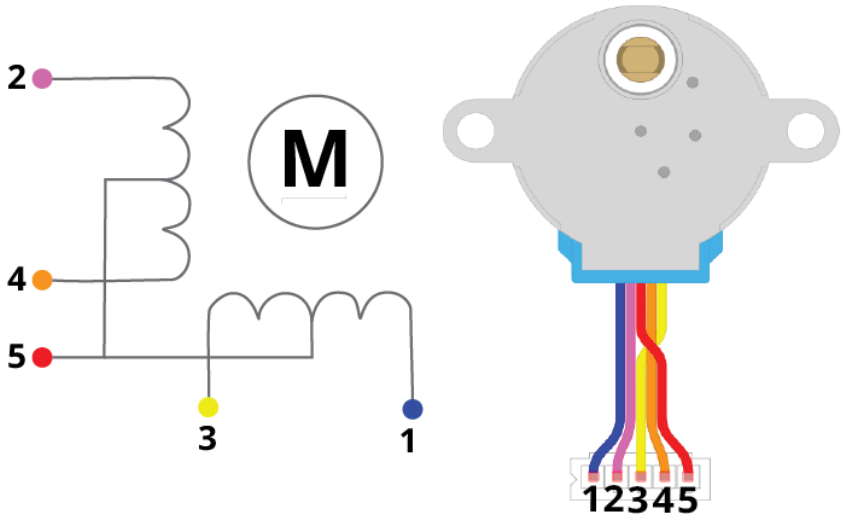


Figura 5 - Bobinas do motor de passo 28BYJ-48. Fonte: site Random Nerd Tutorials.

Figura 6 - Motor de passo 28BYJ-48 5V. Fonte: site Random Nerd Tutorials.

O Módulo Driver ULN2003 foi desenvolvido com a finalidade de acionar cargas indutivas em baixa e média tensão, pois os pinos do ESP não são capazes de acionar este tipo de motor com a tensão e a corrente necessárias. Este módulo controlador possui o circuito integrado ULN2003, que é um driver bastante utilizado em projetos com plataformas microcontroladas.

Figura 7 - Driver ULN2003. Fonte: https://www.casadarobotica.com/robotica/atuadores/drivers/driver-uln2003-para-motor-de-passo

3.3.3 Conexão entre o ESP32 e o Motor de passo 28BYJ-48

Com o driver ULN2003 o processo de conexão e utilização do motor de passo se torna simples. Basta conectar os pinos IN1, IN2, IN3 e IN4 nas GPIOs do ESP32 e conectar o driver em uma fonte de tensão externa. Como ilustrado no diagrama de blocos da Figura 8.

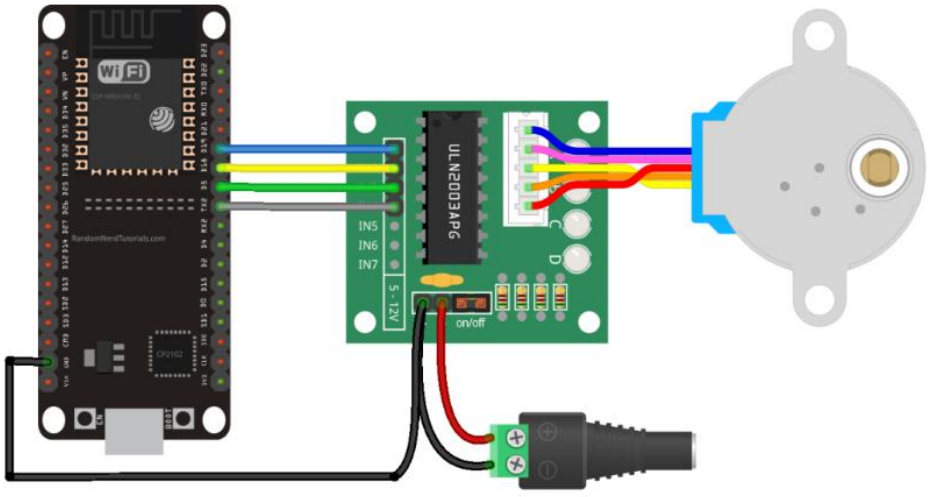


Figura 8 - Diagrama esquemático da conexão entre o ESP32 e o motor a passo. Fonte: site Random Nerd Tutorials.

A Figura 9 mostra a mesma montagem agora utilizando uma protoboard.

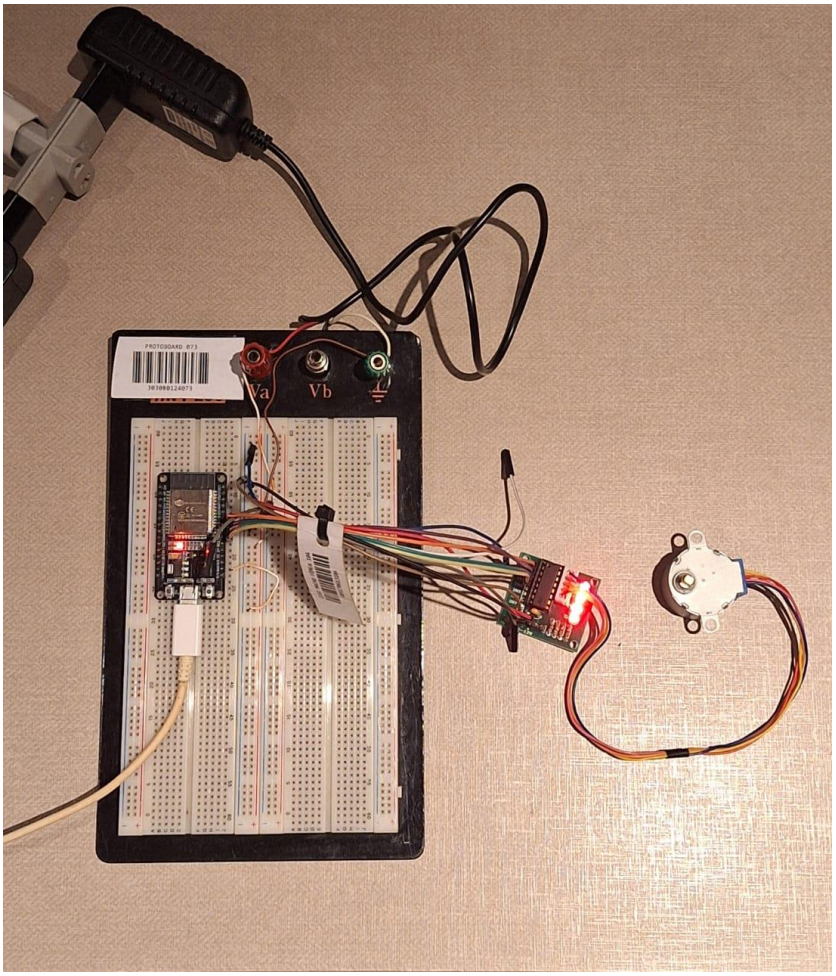


Figura 9 - Circuito montado. Fonte: autor.

3.4 Firmware e interface web

3.4.1 Arduino IDE

Para fazer a programação do *firmware* foi utilizada a IDE do Arduino, uma plataforma cruzada, escrita em linguagem de C e C++. A IDE é utilizada para escrever e fazer *upload* de programas em placas compatíveis com Arduino, e com a ajuda de terceiros, o ESP32 também pode ser utilizado.

3.4.2 Interface *Web*

Para receber os *inputs* do usuário uma página *web* foi criada dentro do código em C++. Na Figura 10 pode-se observar a página *web* criada.

Figura 10 - Página *web*, interface de usuário. Fonte: autor.

O funcionamento da página segue o comportamento demonstrado no fluxograma ilustrado na Figura 11.

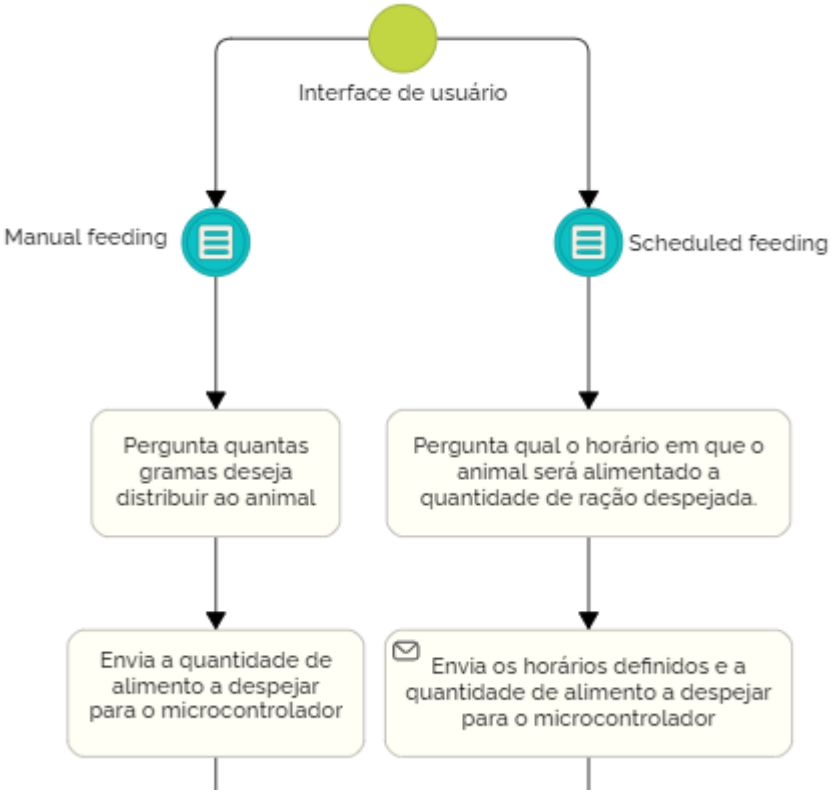


Figura 11 - Fluxograma parcial da página web. Fonte: autor.

A página web pode ser dividida em três partes para uma explicação mais detalhada:

1. ***Manual feeding*:** neste trecho recebe-se a quantidade de comida a distribuir para o animal instantaneamente. A Figura 12 representa a implementação desta parte.

```
<h3>Manual feeding</h3>
<label for="steps">Quantity of food (g):</label>
<input type="number" name="steps">
<input type="submit" value="Feed">
```

Figura 12 - Implementação do *Manual feeding*. Fonte: autor.

2. ***Schedule feeding:*** neste trecho recebemos a quantidade de comida que o proprietário do animal deseja distribuir, bem como o horário no qual o fará. A Figura 13 representa sua a implementação.

```
<h3>Schedule feeding</h3>
<label for="stepsAlarm">Quantity of food (g):</label>
<input type="number" name="stepsAlarm">
<br>
<label for="hour">Hour:</label>
<input type="number" name="hour">
<br>
<label for="minute">Minute:</label>
<input type="number" name="minute">
<br>
<input type="submit" value="Save"><br>
```

Figura 13 - Implementação do Schedule feeding. Fonte: autor.

3. Schedule: realiza a impressão de todos o cronograma de alimentação estabelecido pelo usuário, incluindo um botão *delete* caso haja a necessidade de remover um horário de alimentação indesejado. A Figura 14 descreve a implementação descrita acima. Note que aqui é utilizada uma variável *%MEAL_PLAN_MSG%*, que é a concatenação de todos os horários definidos em uma *string*. Com essa variável podemos mostras todos os horários definidos pelo usuário na página *web*.

```
<h3>Schedule</h3>
<div id="meal-plan-msg">%MEAL_PLAN_MSG%</div>
```

Figura 14 - Implementação da impressão dos horários de alimentação definidos. Fonte: autor

3.4.3 Firmware

Após a criação da página *web* é necessário que o código receba as informações fornecidas pelo usuário e acione o motor para despejar o alimento quando especificado pelo tutor. O fluxograma ilustrado na Figura 15 descreve o processo de comunicação da página *web* com o microcontrolador ESP32 de forma mais clara.

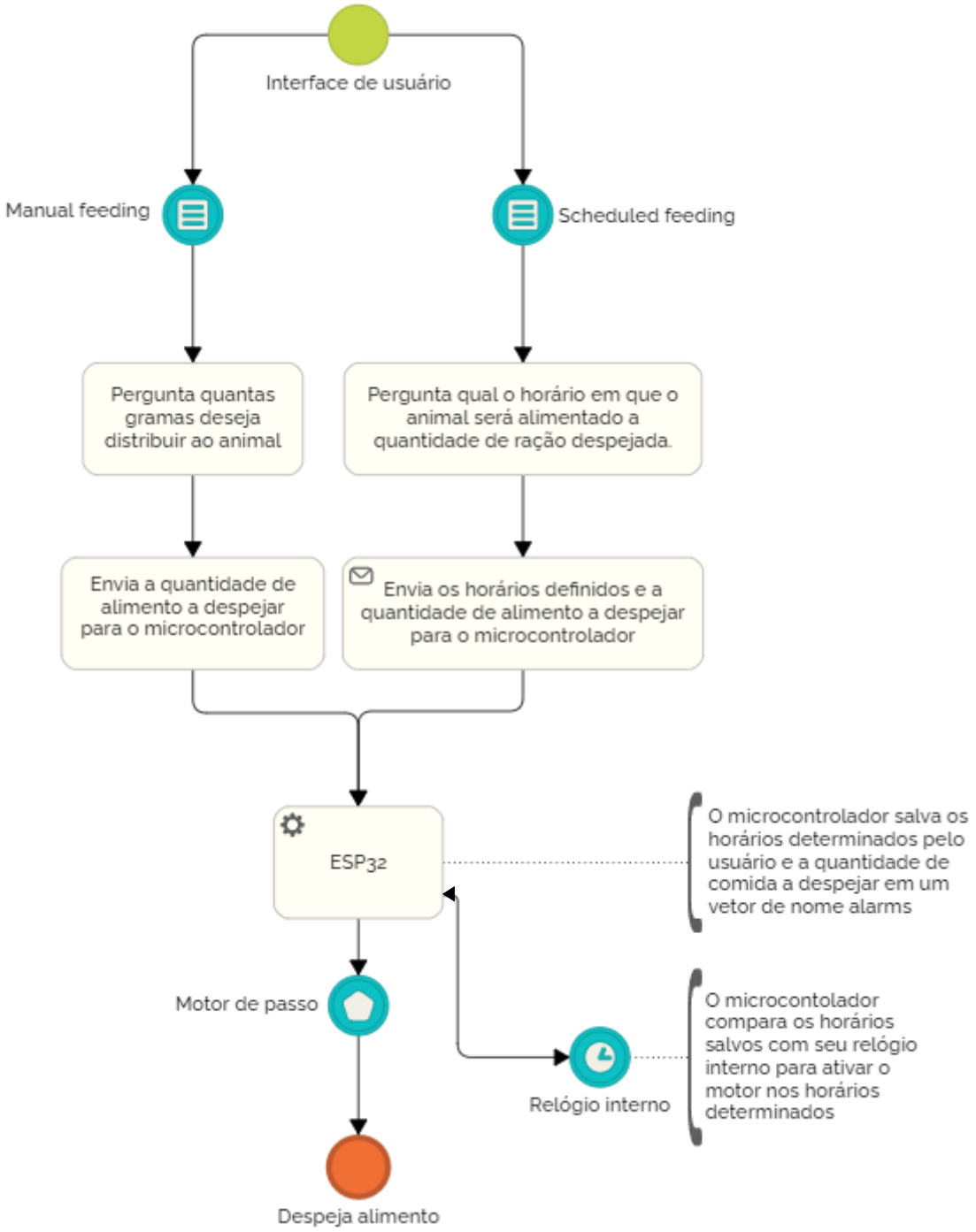


Figura 15 - Fluxograma detalhando o comportamento do protótipo. Fonte: autor.

O comportamento do programa é descrito de forma detalhada a seguir.

A aquisição das informações é feita através de um *POST* que é um método, no contexto deste código, usado para enviar dados de um formulário HTML para o servidor ESP32. Assim que esta informação for recebida as informações passadas pelo usuário são acessadas. Na Figura 16 pode-se observar esta operação.

```
// Handle request (form) (html inputs to use in the code)
server.on("/", HTTP_POST, [](AsyncWebServerRequest *request) {
  int params = request->params();
  bool alarmExists = false;  // Variable to check if an alarm already exists
  bool validTime = true;     // Variable to check if the time is valid (hours less than 24 and minutes less than 60)

  // Retrieve form parameters
  for(int i=0; i<params; i++){
    AsyncWebParameter* p = (AsyncWebParameter*)request->getParam(i); // Add (AsyncWebParameter*)
```

Figura 16 - Operação para receber os inputs do usuário. Fonte: autor.

Após receber as informações do usuário o código faz uma verificação se a operação escolhida foi de adicionar um horário para alimentar o animal. Caso ele deseje salvar um horário para alimentação, uma verificação é feita para saber se o horário já existe no vetor *alarms*, onde são armazenados todos os horários e a quantidade de comida distribuída ao *pet*. Caso o horário exista ele não o adicionará ao vetor. Na Figura 17 podemos observar este comportamento.

```
// If the time is valid, proceed to handle the alarm
if (validTime && manualORschedule == "Schedule") {
  // Check if the alarm already exists
  for (auto &alarm : alarms) {
    if (alarm.hour == hour.toInt() && alarm.minute == minute.toInt()) {
      alarmExists = true;
      break;
    }
  }
  // If the alarm does not exist, add it
  if (!alarmExists) {
    alarms.push_back({hour.toInt(), minute.toInt(), stepsAlarm.toInt(), false});
    Serial.println("Alarm added.");
  } else {
    Serial.println("Alarm already exists!");
  }
}
```

Figura 17 - Verifica se o alarme já existe. Fonte: autor.

Se a operação for alimentação manual, o programa executará apenas o comando de acionamento do motor de passo, como descrito na Figura 18.

```
// Function to feed the pet manually
void manualFeeding(int steps) {
  myStepper.step(steps);
}
```

Figura 18 - Ativação manual do alimentador. Fonte: autor.

Se a operação for de alimentação agendada, depois de definir um vetor para salvar os horários programados, basta criar uma função que verifique se o horário atual corresponde com algum dos horários salvos no vetor *alarms.* Caso o horário atual seja igual a um horário salvo no vetor, o motor de passo será ativado com a mesma função de alimentação manual, como descrito na Figura 19.

```
// Function to feed the pet according to schedule
void scheduleFeeding() {
  time_t now = time(nullptr);
  struct tm *timeinfo = localtime(&now);
  int currentHour = timeinfo->tm_hour;
  int currentMinute = timeinfo->tm_min;

  for (auto &alarm : alarms) {
    if (alarm.hour == currentHour && alarm.minute == currentMinute && !alarm.triggered) {
      manualFeeding(alarm.steps);
      alarm.triggered = true;
      Serial.print("Feeding scheduled at ");
      Serial.print(alarm.hour);
      Serial.print(":");
      Serial.print(alarm.minute);
      Serial.print(" with ");
      Serial.print(alarm.steps);
      Serial.println(" steps.");
    }
  }

  // Reset alarm triggers at midnight
  if (currentHour == 0 && currentMinute == 0) {
    for (auto &alarm : alarms) {
      alarm.triggered = false;
    }
  }
}
```

Figura 19 - Ativação programada do alimentador. Fonte: autor.

3.4.4 Análise de custo do projeto

Analisando o custo do projeto em relação as opções disponíveis no mercado obtêm-se os seguintes resultados:

Custos do Projeto:

- ESP32: R$ 40,00
- Motor de passo (28BYJ-48): R$ 20,00
- Filamento (500g): aproximadamente R$ 50,00 (considerando um preço médio de R$ 100,00 por kg de filamento)
- Garrafa PET: R$ 1,20
- Jumpers (7 unidades): R$ 2,00
- Fonte de 5V: aproximadamente R$ 15,00

Total: R$ 128,20

Comparação com Dispositivos Comercializados:

Os preços são baseados em pesquisas de mercado e podem variar:

- PetSafe Automatic Feeder: cerca de R$ 700,00
- Xiaomi Smart Pet Feeder: aproximadamente R$ 900,00
- PetKit Smart Feeder: aproximadamente R$ 1.200,00
- Alimentador automático básico (sem conectividade *smart*): cerca de R$ 300,00 a R$ 500,00

Os preços variam entre R$ 300,00 à R$ 1.200,00.

CONCLUSÃO

Com este projeto, nota-se que o alimentador automático se torna uma opção para que se tente fazer uma alimentação saudável dos animais para proporcionar uma dieta mais balanceada aos *pets*, e também proporcionar um dispositivo que torna uma tarefa diária muito menos incômoda.

Com os dados sobre o custo do projeto mencionados anteriormente, conclui-se que o projeto é significativamente mais econômico do que os dispositivos básicos comercializados, especialmente aqueles com funcionalidades inteligentes.

Este protótipo é apenas uma versão inicial de um alimentador automático, podendo conter diversas outras funções que não serão abordadas neste projeto, como câmeras, *buzzer* para avisar que o alimento está sendo distribuído, um auto falante e muitas outras funcionalidades, além é claro, de mecanismos que tornam o dispositivo mais robusto e tolerante a falhas.

Com este projeto foi possível integrar diversas áreas de conhecimentos e competências desenvolvidas durante o curso de engenharia de computação, demonstrando o caráter multidisciplinar do curso a seus egressos.

REFERÊNCIAS

[1] SEBRAE. **Crescimento do mercado pet é oportunidade de negócio**. Disponível em: https://sebrae.com.br/sites/PortalSebrae/ufs/al/artigos/crescimento-do-mercado-pet-e-oportunidade-de-negocio,021731b7fe057810VgnVCM1000001b00320aRCRD. Acesso em: 29 jun. 2024.

[2] PETLOVE. **Nutrição de cães e gatos: Guia completo**. Disponível em: https://www.petlove.com.br/dicas/nutricao-de-caes-e-gatos-guia-completo#:~:text=Quando%20nossos%20pets%20comem%20o,o%20que%20eles%20comem%20diariamente. Acesso em: 29 jun. 2024.

[3] **AMERICAN VETERINARY MEDICAL ASSOCIATION*. Study finds overweight dogs live shorter lives***. Disponível em: https://www.avma.org/javma-news/2019-03-01/study-finds-overweight-dogs-live-shorter-lives. Acesso em: 1 jul. 2024.

[4] CNX SOFTWARE. **ESP8266 and ESP32 differences in one single table.** Disponível em: https://www.cnx-software.com/2016/03/25/esp8266-and-esp32-differences-in-one-single-table/. Acesso em: 1 jul. 2024.

[5] **Modelo 3D do projeto**. Disponível em: https://smartsolutions4home.com/ss4h-pf-pet-feeder/. Acesso em: 25 mar. 2024.

[6] RANDOM NERD TUTORIALS. Guia de como usar o esp32 com o motor de passo 28BYJ-48. Disponível em: https://randomnerdtutorials.com/esp32-stepper-motor-28byj-48-uln2003/. Acesso em: 27 jun. 2024.

[7] RANDOM NERD TUTORIALS. Guia de como criar uma página HTML. Disponível em: https://randomnerdtutorials.com/stepper-motor-esp32-web-server/. Acesso em: 26 jun. 2024.

[8] **Site para criar fluxogramas.** Disponível em: https://app.heflo.com/. Acesso em: 1 jul. 2024.

[9] CIRCUITSTATE. **DOIT ESP32 DevKit V1 Wi-Fi Development Board – Pinout Diagram & Arduino Reference.** Disponível em: https://www.circuitstate.com/pinouts/doit-esp32-devkit-v1-wifi-development-board-pinout-diagram-and-reference/. Acesso em: 1 jul. 2024.

[10] ESPRESSIF. **About Arduino ESP32.** Disponível em: https://docs.espressif.com/projects/arduino-esp32/en/latest/getting_started.html. Acesso em: 28 jun. 2024.

ANEXO A

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <Stepper.h>
#include <time.h>
#include <vector>
#include <algorithm> // for std::sort

// Stepper Motor Settings
const int stepsPerRevolution = 2048;  // change this to fit the number of
steps per revolution
#define IN1 19
#define IN2 18
#define IN3 5
#define IN4 17
Stepper myStepper(stepsPerRevolution, IN1, IN3, IN2, IN4);

// Replace with your network credentials
const char* ssid = "ThiagoW";
const char* password = "thiago123";

// Create AsyncWebServer object on port 80
AsyncWebServer server(80);

// Search for parameters in HTTP POST request
const char* PARAM_MANUAL_FEED = "steps";
const char* PARAM_SCHEDULE_FEED = "stepsAlarm";
const char* PARAM_SCHEDULE_HOUR = "hour";
const char* PARAM_SCHEDULE_MINUTE = "minute";
const char* PARAM_DELETE_ALARM = "deleteAlarm";

// Variables to save values from HTML form
String manualORschedule;
String steps;
String stepsAlarm;
String hour;
String minute;

// Alarm structure
struct Alarm {
  int hour;
  int minute;
  int steps;
  bool triggered;  // To track if the alarm has been triggered
```

```
};

std::vector<Alarm> alarms;

// Variable to detect new request
bool newRequest = false;

// HTML to build the web page
const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html>
<head>
  <title>Stepper Motor</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script>
    function updateTime() {
      var now = new Date();
      var hours = now.getHours().toString().padStart(2, '0');
      var minutes = now.getMinutes().toString().padStart(2, '0');
      var seconds = now.getSeconds().toString().padStart(2, '0');
      document.getElementById('clock').innerHTML = hours + ':' + minutes + ':'
+ seconds;
    }
    setInterval(updateTime, 1000);
  </script>
</head>
<body onload="updateTime()">
  <h1>PET FEEDER</h1>
  <div id="clock"></div>
  <form action="/" method="POST">
    <h3>Manual feeding</h3>
    <label for="steps">Quantity of food (g):</label>
    <input type="number" name="steps">
    <input type="submit" value="Feed">
    <h3>Schedule feeding</h3>
    <label for="stepsAlarm">Quantity of food (g):</label>
    <input type="number" name="stepsAlarm">
    <br>
    <label for="hour">Hour:</label>
    <input type="number" name="hour">
    <br>
    <label for="minute">Minute:</label>
    <input type="number" name="minute">
    <br>
    <input type="submit" value="Save"><br>
  </form>
  <br>
  <h3>Schedule</h3>
  <div id="meal-plan-msg">%MEAL_PLAN_MSG%</div>
```

```
</body>
</html>
)rawliteral";

// Initialize WiFi
void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi ..");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
  }
  Serial.println(WiFi.localIP());
}

// Initialize and sync time with timezone
void initTime() {
  configTime(-3 * 3600, 0, "pool.ntp.org", "time.nist.gov");
  Serial.print("Waiting for time");
  while (!time(nullptr)) {
    Serial.print(".");
    delay(1000);
  }
  Serial.println("");
  Serial.println("Time initialized!");
}

// Function to add leading zeros
String padZero(int num) {
  if (num < 10) {
    return "0" + String(num);
  }
  return String(num);
}

// Function to generate HTML for meal plan
String getMealPlanMsg() {
  String msg;

  // Sort alarms by time (hour and minute)
  std::sort(alarms.begin(), alarms.end(), [](const Alarm &a, const Alarm &b) {
    if (a.hour == b.hour) {
      return a.minute < b.minute;
    }
    return a.hour < b.hour;
  });

  // Concatenate alarms into a string to send it to HTML Scheduled alarms
```

```
  for (size_t i = 0; i < alarms.size(); ++i) {
    int grams = (alarms[i].steps * 10) / 1024; // Convert steps back to grams
    msg += padZero(alarms[i].hour) + ":" + padZero(alarms[i].minute) + " - " +
String(grams) + "g <form action='/delete' method='POST'><button type='submit'
name='deleteAlarm' value='" + String(i) + "'>Delete</button></form><br>";
  }
  return msg;
}

// Function to convert grams to steps
int gramsToSteps(int grams) {
  return (grams * 1024) / 10;
}

void setup() {
  Serial.begin(115200);

  initWiFi();
  initTime();

  myStepper.setSpeed(10);

  // Web Server Root URL
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    String html = String(index_html);
    html.replace("%MEAL_PLAN_MSG%", getMealPlanMsg());
    request->send(200, "text/html", html);
  });

  // Handle request (form) (html inputs to use in the code)
  server.on("/", HTTP_POST, [](AsyncWebServerRequest *request) {
    int params = request->params();
    bool alarmExists = false;  // Variable to check if an alarm already exists
    bool validTime = true;     // Variable to check if the time is valid
(hours less than 24 and minutes less than 60)

    // Retrieve form parameters
    for(int i=0; i<params; i++){
      AsyncWebParameter* p = (AsyncWebParameter*)request->getParam(i); // Add
(AsyncWebParameter*)
      if(p->isPost()){
        // HTTP POST input1 value (steps)
        if (p->name() == PARAM_MANUAL_FEED) {
          steps = p->value().c_str();
          int stepsInt = gramsToSteps(steps.toInt()); // Convert grams to
steps
          Serial.print("Quantity of food to feed: ");
          Serial.println(stepsInt);
          if(stepsInt > 0){
```

```
            manualORschedule = "Manual";
            steps = String(stepsInt); // Update steps variable with the
converted value
          }
          Serial.println(manualORschedule);
        }
        // HTTP POST input2 value (stepsAlarm)
        if (p->name() == PARAM_SCHEDULE_FEED) {
          stepsAlarm = p->value().c_str();
          int stepsAlarmInt = gramsToSteps(stepsAlarm.toInt()); // Convert
grams to steps
          Serial.print("Quantity of food in the alarm set to: ");
          Serial.println(stepsAlarmInt);
          if(stepsAlarmInt > 0){
            manualORschedule = "Schedule";
            stepsAlarm = String(stepsAlarmInt); // Update stepsAlarm variable
with the converted value
          }
          Serial.println(manualORschedule);
        }
        // HTTP POST input3 value (hour)
        if (p->name() == PARAM_SCHEDULE_HOUR) {
          hour = p->value().c_str();
          Serial.print("Hour set to: ");
          Serial.println(hour);
          // Check if hour is valid
          int hourInt = hour.toInt();
          if (hourInt < 0 || hourInt > 23) {
            validTime = false;
            Serial.println("Invalid hour!");
          }
        }
        // HTTP POST input4 value (minute)
        if (p->name() == PARAM_SCHEDULE_MINUTE) {
          minute = p->value().c_str();
          Serial.print("Minute set to: ");
          Serial.println(minute);
          // Check if minute is valid
          int minuteInt = minute.toInt();
          if (minuteInt < 0 || minuteInt > 59) {
            validTime = false;
            Serial.println("Invalid minute!");
          }
        }
      }
    }

    // If the time is valid, proceed to handle the alarm
    if (validTime && manualORschedule == "Schedule") {
```

```
      // Check if the alarm already exists
      for (auto &alarm : alarms) {
        if (alarm.hour == hour.toInt() && alarm.minute == minute.toInt()) {
          alarmExists = true;
          break;
        }
      }
      // If the alarm does not exist, add it
      if (!alarmExists) {
        alarms.push_back({hour.toInt(), minute.toInt(), stepsAlarm.toInt(),
false});
        Serial.println("Alarm added.");
      } else {
        Serial.println("Alarm already exists!");
      }
    }

    newRequest = true;
    String html = String(index_html);
    html.replace("%MEAL_PLAN_MSG%", getMealPlanMsg());
    request->send(200, "text/html", html);
  });

  // Handle request to delete alarm
  server.on("/delete", HTTP_POST, [](AsyncWebServerRequest *request) {
    if (request->hasParam(PARAM_DELETE_ALARM, true)) {
      int index = request->getParam(PARAM_DELETE_ALARM, true)-
>value().toInt();
      if (index >= 0 && index < alarms.size()) {
        alarms.erase(alarms.begin() + index);
        Serial.println("Alarm deleted.");
      }
    }
    String html = String(index_html);
    html.replace("%MEAL_PLAN_MSG%", getMealPlanMsg());
    request->send(200, "text/html", html);
  });

  server.begin();
}

// Function to feed the pet manually
void manualFeeding(int steps) {
  myStepper.step(steps);
}

// Function to feed the pet according to schedule
void scheduleFeeding() {
  time_t now = time(nullptr);
```

```
  struct tm *timeinfo = localtime(&now);
  int currentHour = timeinfo->tm_hour;
  int currentMinute = timeinfo->tm_min;

  for (auto &alarm : alarms) {
    if (alarm.hour == currentHour && alarm.minute == currentMinute &&
!alarm.triggered) {
      manualFeeding(alarm.steps);
      alarm.triggered = true;
      Serial.print("Feeding scheduled at ");
      Serial.print(alarm.hour);
      Serial.print(":");
      Serial.print(alarm.minute);
      Serial.print(" with ");
      Serial.print(alarm.steps);
      Serial.println(" steps.");
    }
  }

  // Reset alarm triggers at midnight
  if (currentHour == 0 && currentMinute == 0) {
    for (auto &alarm : alarms) {
      alarm.triggered = false;
    }
  }
}

void loop() {
  // Handle new request
  if (newRequest) {
    if (manualORschedule == "Manual" && steps.toInt() > 0) {
      manualFeeding(steps.toInt());
      steps = "";
      manualORschedule = "";
    }
    newRequest = false;
  }

  // Handle scheduled feeding
  scheduleFeeding();
}
```