UNICAMP

ES770 - Laboratório de Sistemas Digitais

Aula 14 - Corrida Final

Laura Marchione	*156169*
Rafael Sol Santos Martins	*160234*
Victor Cintra Santos	*157461*

Sumário

Introdução

Este relatório tem como intuito documentar e ilustrar todo o planejamento e desenvolvimento de um veículo autônomo seguidor de linha, para a disciplina de Laboratório de Sistemas Digitais (ES770B - 2019S2), lecionada pelo Prof. Dr. Euripedes Guilherme de Oliveira Nobrega junto com o PED Vinicius Benites Bastos.

O produto será um carrinho capaz de seguir um circuito previamente marcado, capaz de se auto controlar e determinar o final do percurso. Aqui, documentamos o projeto mecânico, elétrico e a lógica do código desenvolvido para o sistema embarcado.

Projeto Mecânico

Para desenvolvermos o projeto, compramos partes de um kit de desenvolvimento para embarcados já existente no mercado. Tal kit vem com o chassi, rodas e motores do carrinho, como na figura abaixo.

Imagem 01 - Exemplo kit carrinho embarcado

Os componentes mecânicos utilizados no veículo, foram:

- 2x Motores DC (TT Motor) 6V
- Chassi
- 2x Rodas Laterais
- Roda dianteira
- Suporte para pilhas

Além dos principais componentes acima, não detalharemos no relatório os parafusos e outros suportes utilizados no projeto. Outros componentes serão detalhados na parte eletrônica do relatório. Para as ilustrações mecânicas e detalhamento de dimensões, utilizamos o software online OnShape.

Motor DC

O motor utilizado no projeto foi um de 6V, com uma caixa de redução, conhecido como TT Motor. Tal componente já vem acoplado a um sistema de redução que aumenta o torque do motor, incluindo também o eixo de comunicação para acoplar as rodas e encoder do tacômetro.
Abaixo, segue o detalhamento dos motores, bem como suas dimensões. Sua funcionalidade eletrônica será detalhada em outra seção do documento.

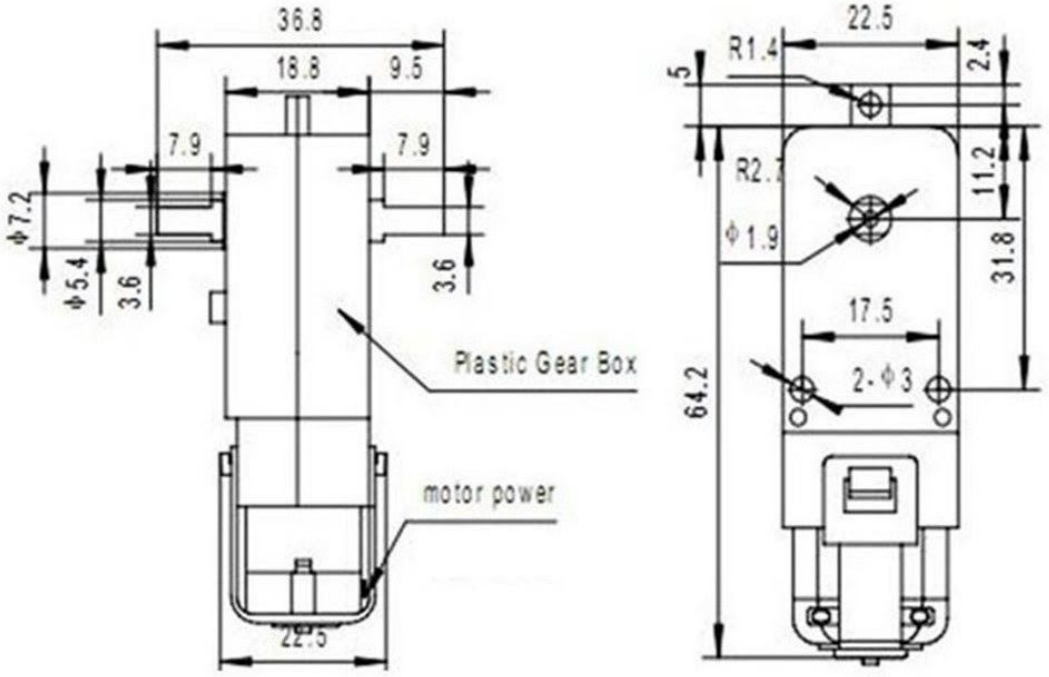


Imagem 02 - Detalhamento motor DC

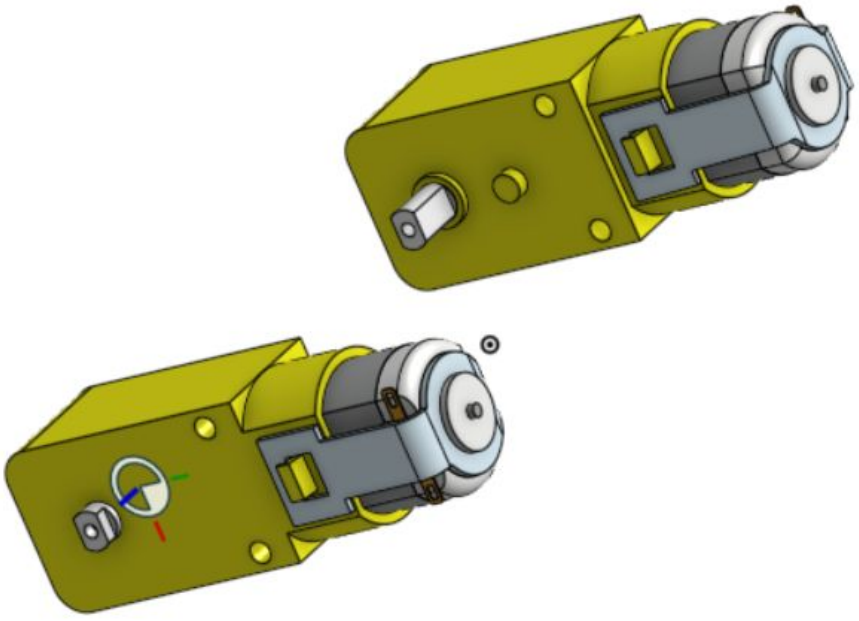


Imagem 03 - Visão motores e caixa redutora

Rodas

Nos motores traseiros são acopladas rodas de borracha, para melhor tração do carrinho, com as seguintes dimensões:

- 66mm de diâmetro exterior
- 51,8mm de diâmetro interior
- 26,6mm de largura
- 3.66mm largura do furo do eixo
- 5,3mm comprimento do furo do eixo

Abaixo, segue a imagem da roda de referência:

Imagem 04 - Roda traseira

Na parte dianteira, incluímos uma roda simples, sem tração, capaz de rotacionar para permitir o carro fazer curvas, chamada de roda rodízio giratória. Abaixo, segue a ilustração.

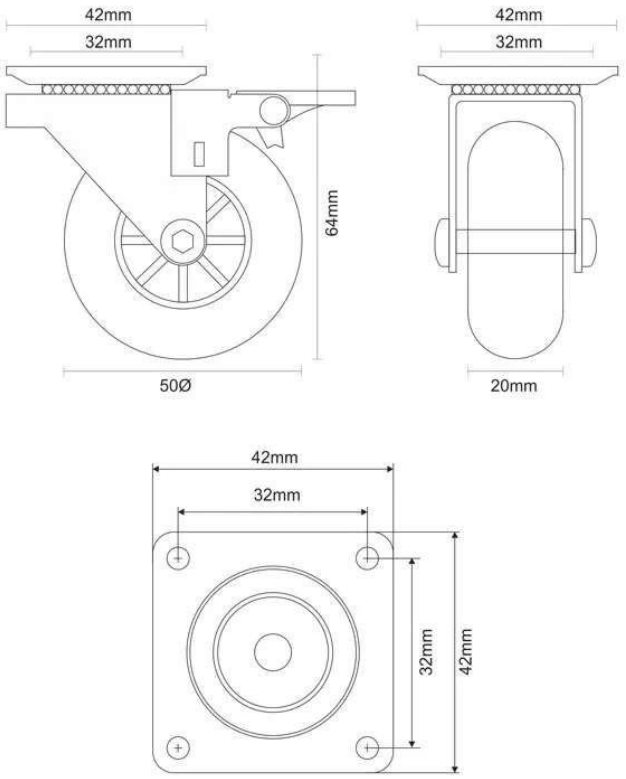


Imagem 05 - Roda dianteira

Chassi

O chassi utilizado já vem com furos básicos para acoplar os elementos principais do carrinho, como indica a figura abaixo. Fizemos mais alguns furos para incluir outros componentes eletrônicos, como a ponte H que posicionamos em baixo do carrinho entre os motores e os sensores infravermelhos, que pode ser melhor visto nas imagens finais deste relatório.

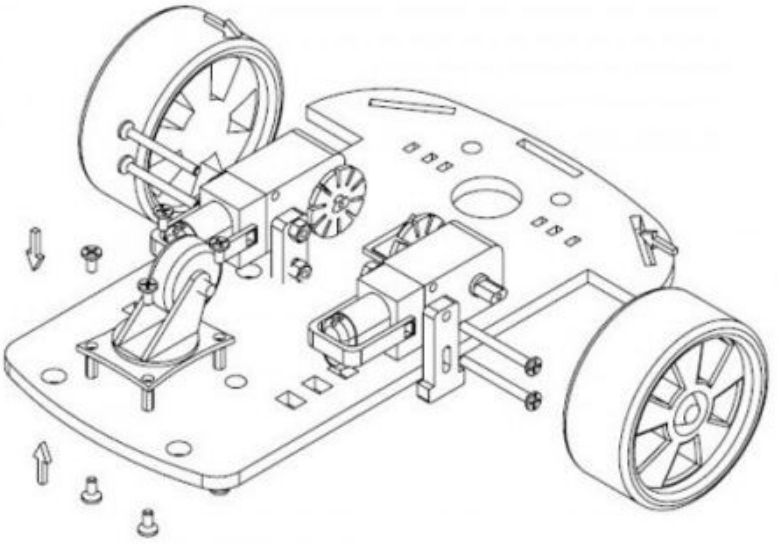


Imagem 06 - Montagem básica do chassi

Projeto Eletrônico

Nesta seção, trazemos os diagramas eletrônicos projetados para o desenvolvimento do projeto. Incluímos referências dos componentes utilizados e as ligações realizadas. O projeto foi desenvolvido numa placa de prototipagem (protoboard) com os seguintes componentes: Para tal veículo autônomo, temos os seguintes elementos:

- Placa controladora ESP32
- Módulo Ponte H L298N
- 2x Motores DC (TT Motor) 6V
- Sensor Infravermelho BFD-1000 (5x TCRT-5000)
- Alimentação externa 6V

Abaixo, ilustramos as principais esquemáticas e informações de cada componente, bem como o diagrama completo para a montagem do projeto.

ESP32

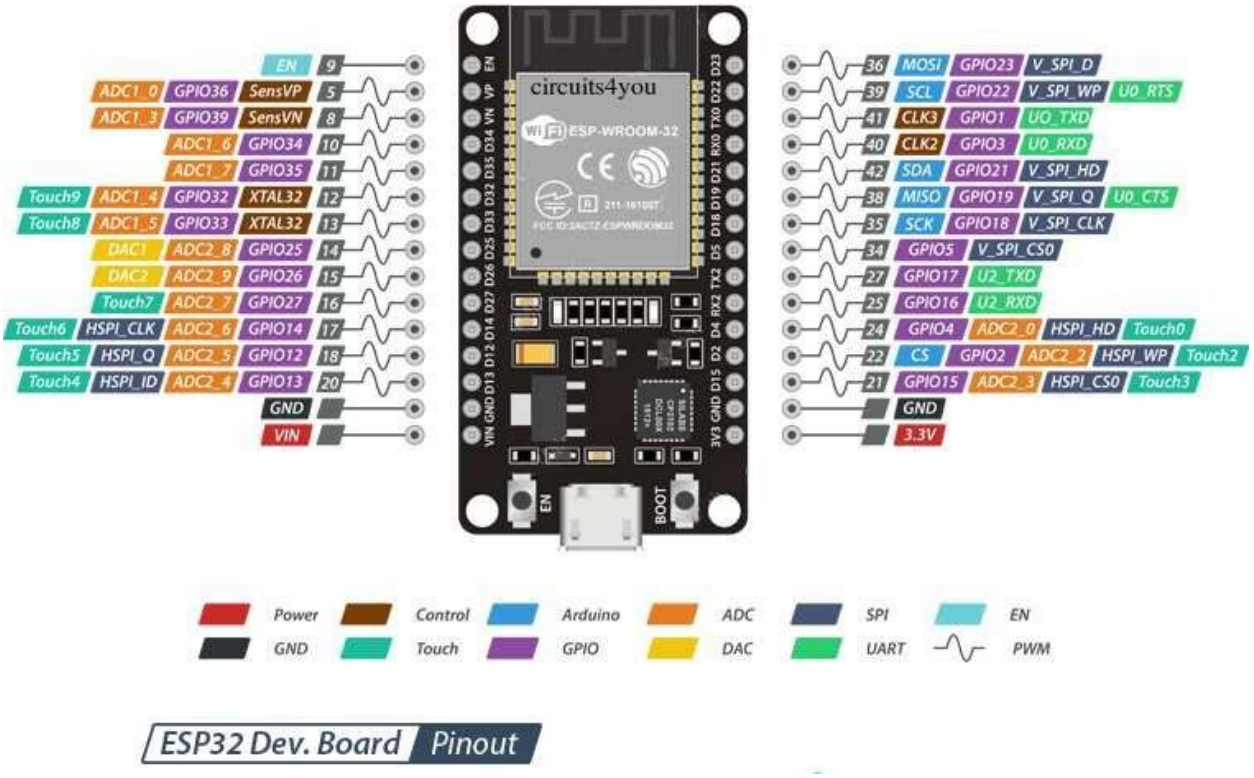


Imagem 07 - Pinout ESP32 Dev Kit

A imagem acima indica os pinos e suas principais características do kit de desenvolvimento do módulo ESP32 utilizado para o carrinho. Tal placa possui pinos padrões de GPIO bem como saídas PWM para o controle dos motores DC. Sua função é de receber as leituras dos sensores infravermelhos para controlar a rotação dos motores do carrinho.

Ponte H L298N

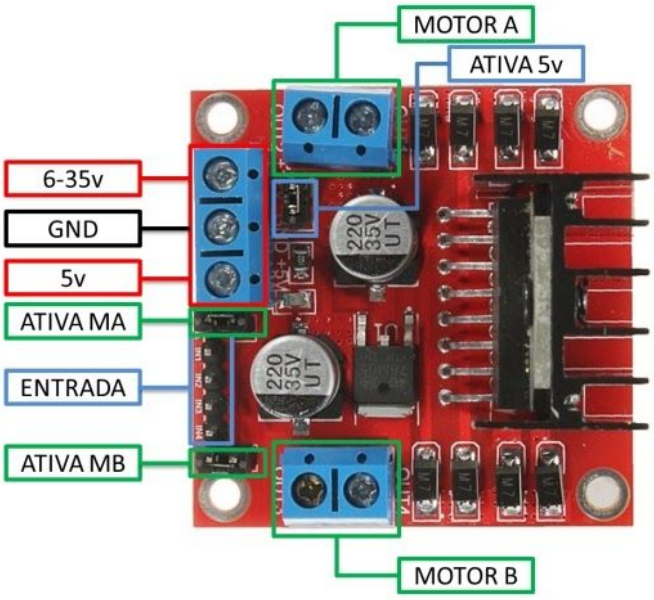


Imagem 08 - Ponte H

Tal módulo possui dois circuitos isolados de transistores responsáveis pelo chaveamento entre o sinal de entrada de controle e a fonte de alimentação externa. Pelos pinos de *Entrada* e *Ativa MA/MB* controlamos por um sinal PWM o sentido e intensidade de rotação dos motores, que são ligados aos pinos *Motor A/B* . O pino *Ativa 5V* é deixado em curto com um jumper, para que a ponte H seja alimentada por uma fonte externa, que é ligada ao *GND* e *6V*.

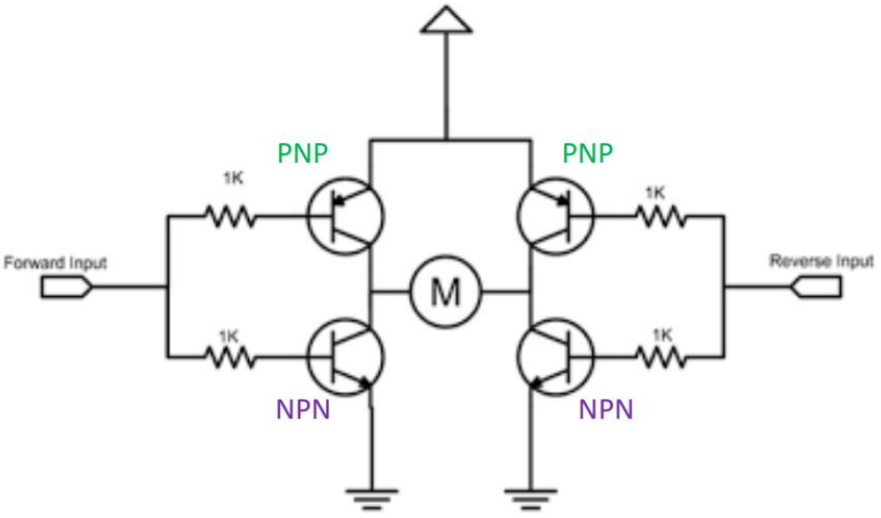


Imagem 09 - Circuito interno Ponte H

Nos pinos Motor A e Motor B temos 2 saídas para cada motor, dependendo da combinação dos sinais aplicados nos pares de Entradas temos o seguinte comportamento nos motores, onde IN1 e IN2 se referem a dupla de controle do Motor A; IN3 e IN4 do Motor B.

MOTOR	IN1	IN2
HORÁRIO	5v	GND
ANTI-HORÁRIO	GND	5v
PONTO MORTO	GND	GND
FREIO	5v	5v

Motores DC (TT Motor)

Os motores DC são dois motores simples, de 6V, acoplados a uma caixa de redução, que serão alimentados por uma fonte externa de 6V e controlados pelo módulo ponte H. Os motores possuem uma tensão de operação de 3 à 6V fornecendo uma rotação de 200 rpm. A esquemática do motor será melhor tratada no diagrama mecânico.

Sensor Infravermelho

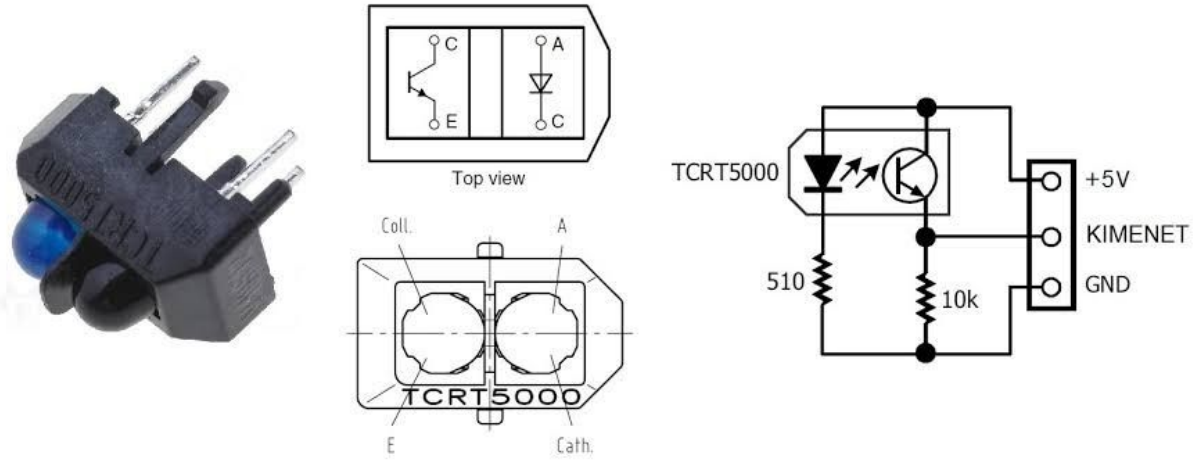


Imagem 10 - Diagrama TCRT5000

Acima é indicado o diagrama do sensor infravermelho básico utilizado. Através de um diodo emissor infravermelho e um transistor receptor é possível detectar a posição do carrinho ao longo de um percurso. Tal sensor foi acoplado com outros num módulo com 6 sensores infravermelhos e um switch para detectar colisões, chamado de BFD-1000, tendo o mesmo funcionamento.

Conexões Placa

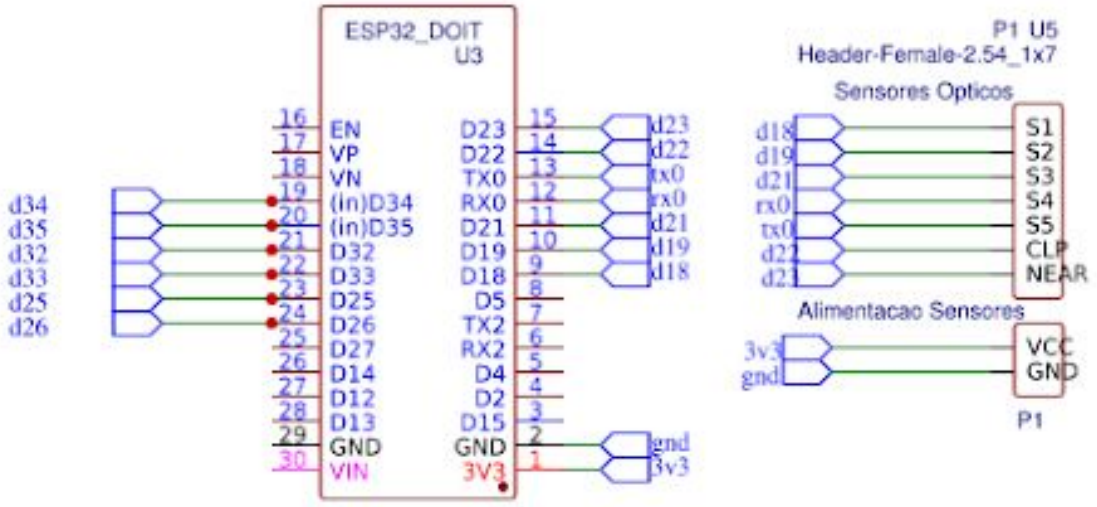


Imagem 11 - Conexões com controlador

Diagrama Eletrônico Completo

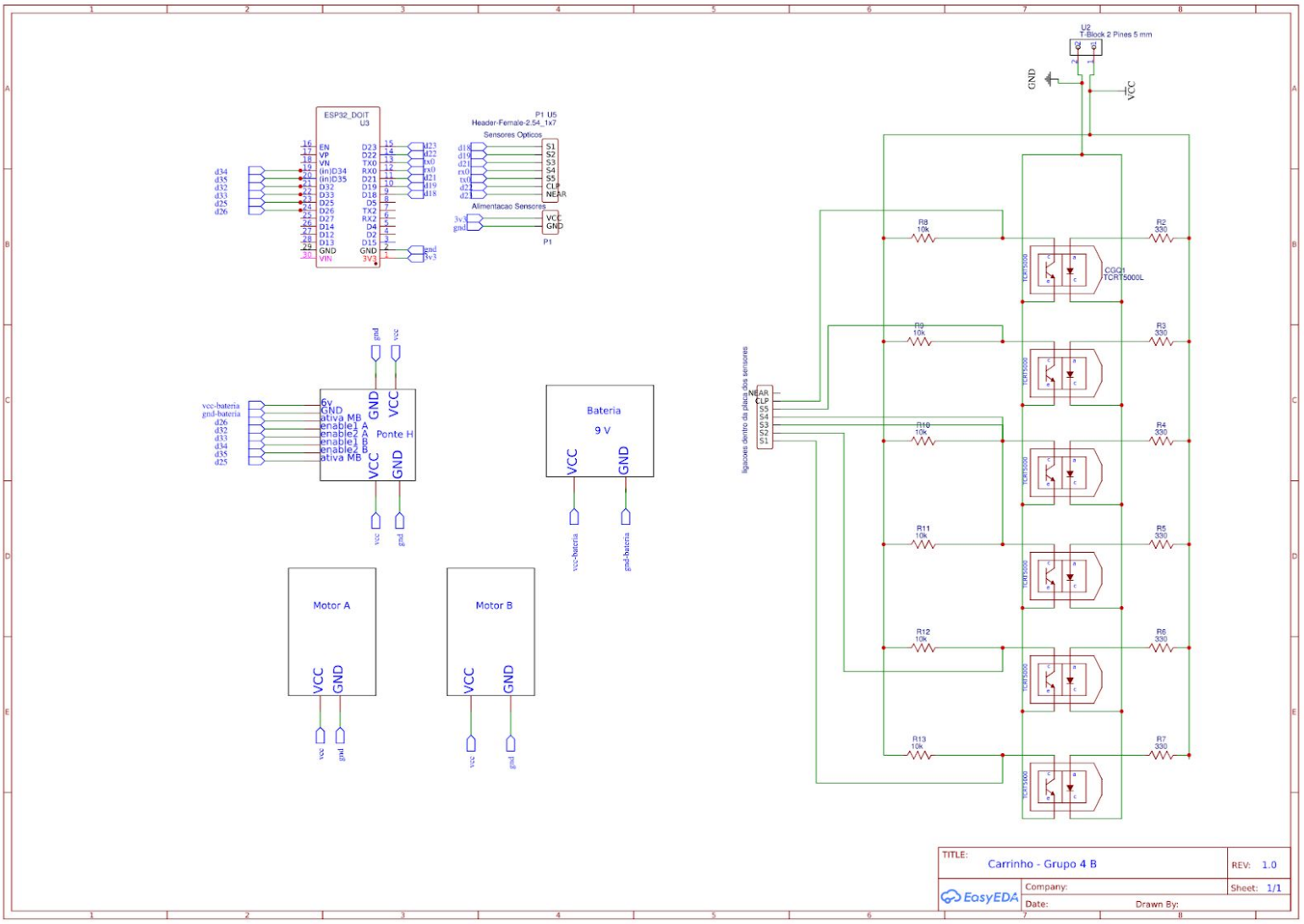
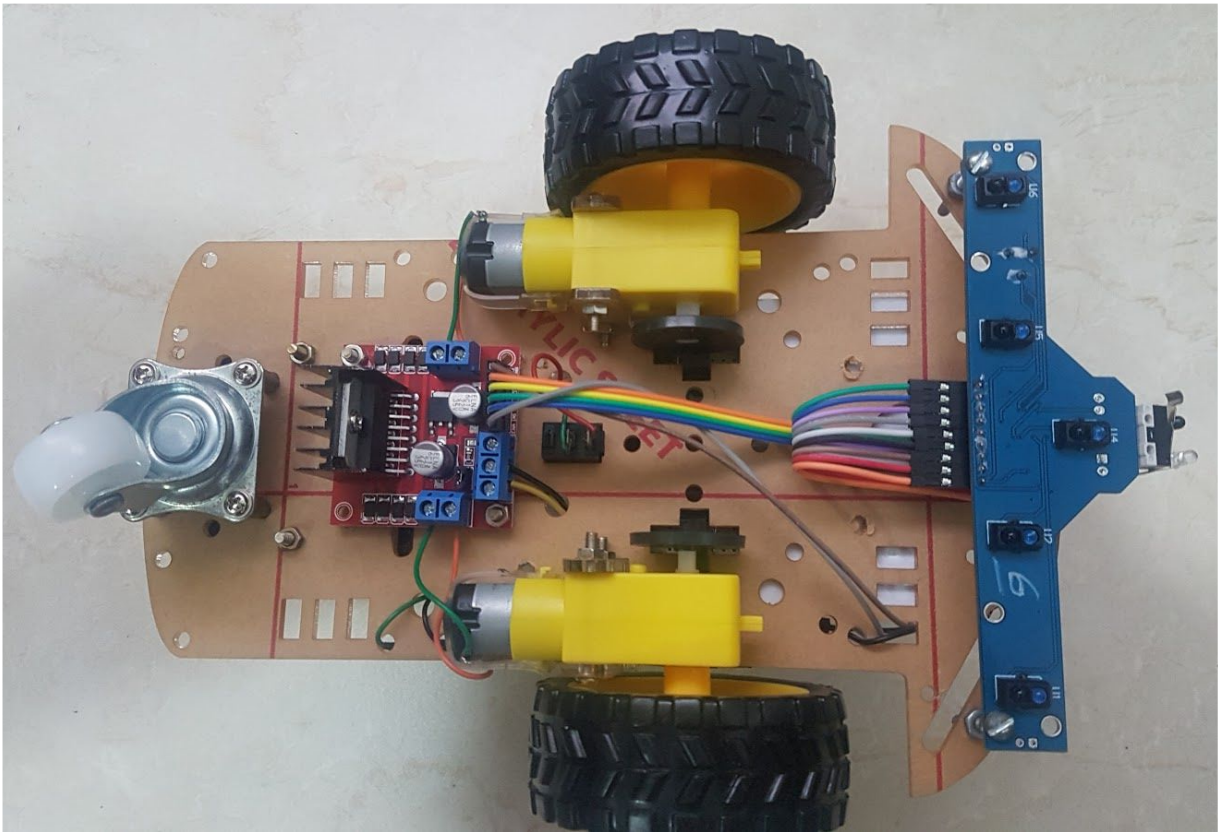
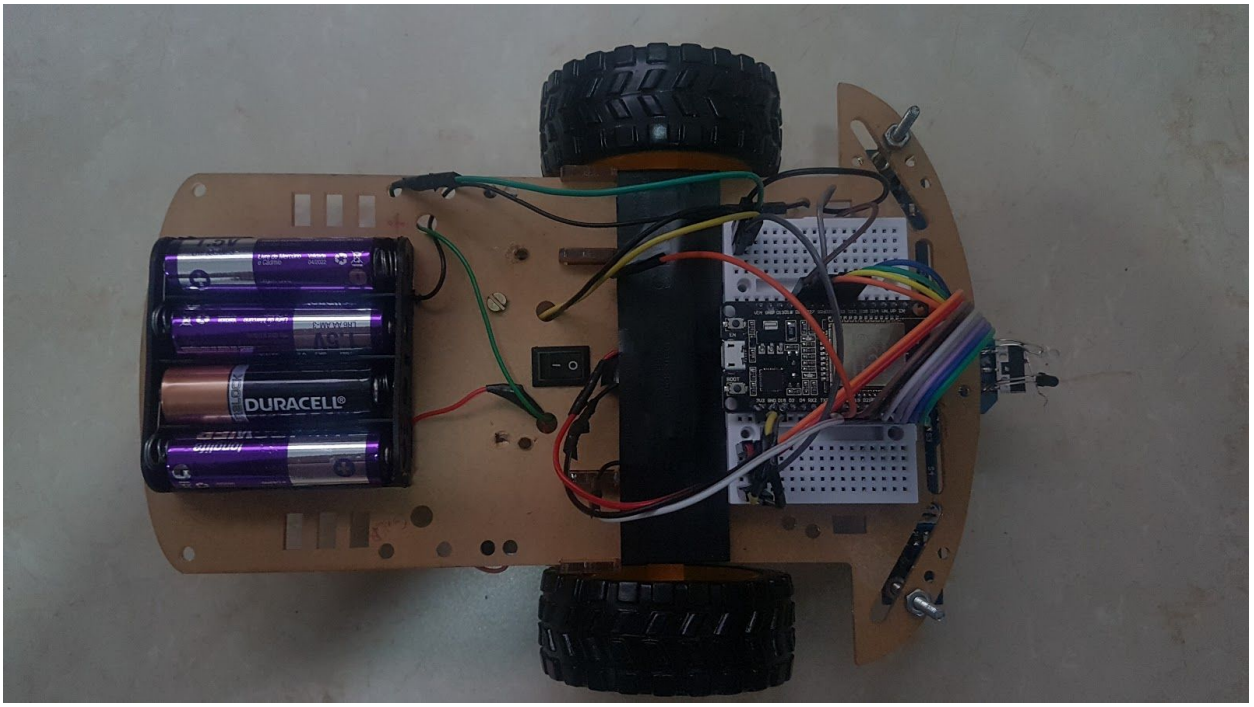


Imagem 12 - Diagrama de conexões completo

Montagem Final

Como podemos ver nas imagens a seguir, temos a montagem final do carrinho com todas as peças mecânicas e eletrônicas. Nelas, podemos ver com maior facilidade posicionamento da ponte H e sensores infravermelhos. A ponte H foi posicionada na parte inferior para facilitar na comunicação com os motores. Já os sensores foram colocados para melhor detecção da pista, na parte frontal do carrinho.





Principais Diagramas de Projeto

Nessa seção, trazemos os diagramas UML para o desenvolvimento do projeto. Para tal, temos os seguintes diagramas:

- Casos de uso
- Atividade
- Comunicações
- Classes
- Sequência
- Máquina de estados

Diagrama de Casos de Uso

Descrição

Esse caso de uso descreve o comportamento de um carrinho seguidor de linha e a interação com o usuário. O usuário liga o carrinho e o posiciona em cima do início do percurso a ser seguido e o veículo começa a percorrê-lo.

Fluxo

- O carrinho deve monitorar sua trajetória
- A velocidade dos motores deve ser calculada pelo controlador considerando a leitura dos sensores infravermelhos
- O final do percurso deve ser detectado pelo carrinho

Requerimentos

- Ser um veículo autônomo

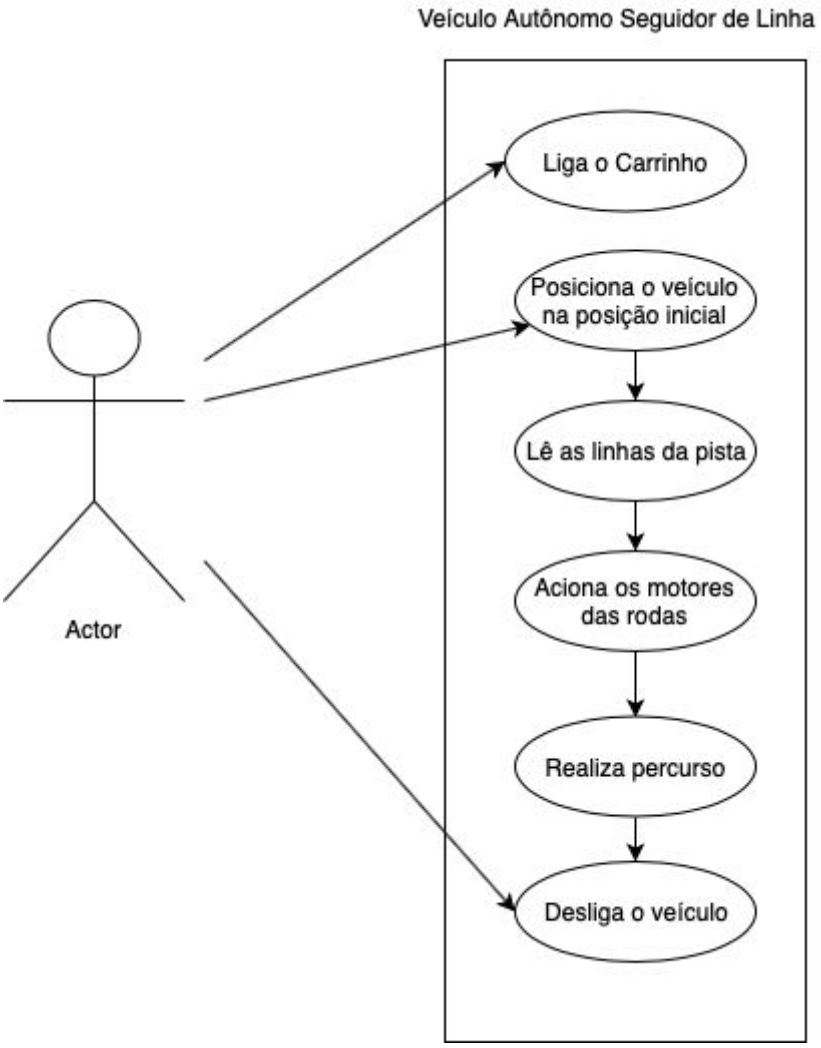


Imagem 1 - Diagrama de Caso de Uso

Diagrama de Atividades

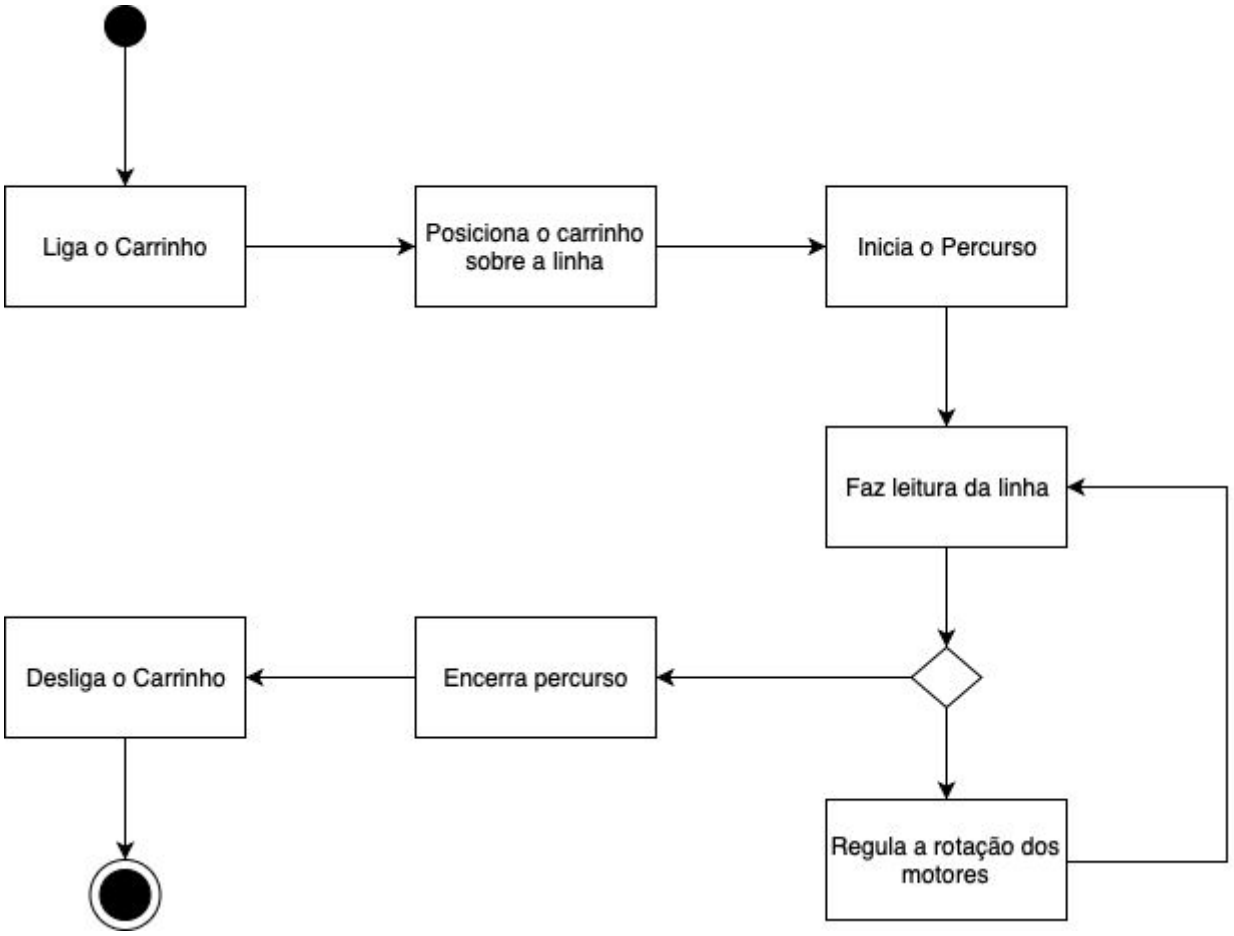


Imagem 2 - Diagrama de Atividades

Diagrama de Comunicação

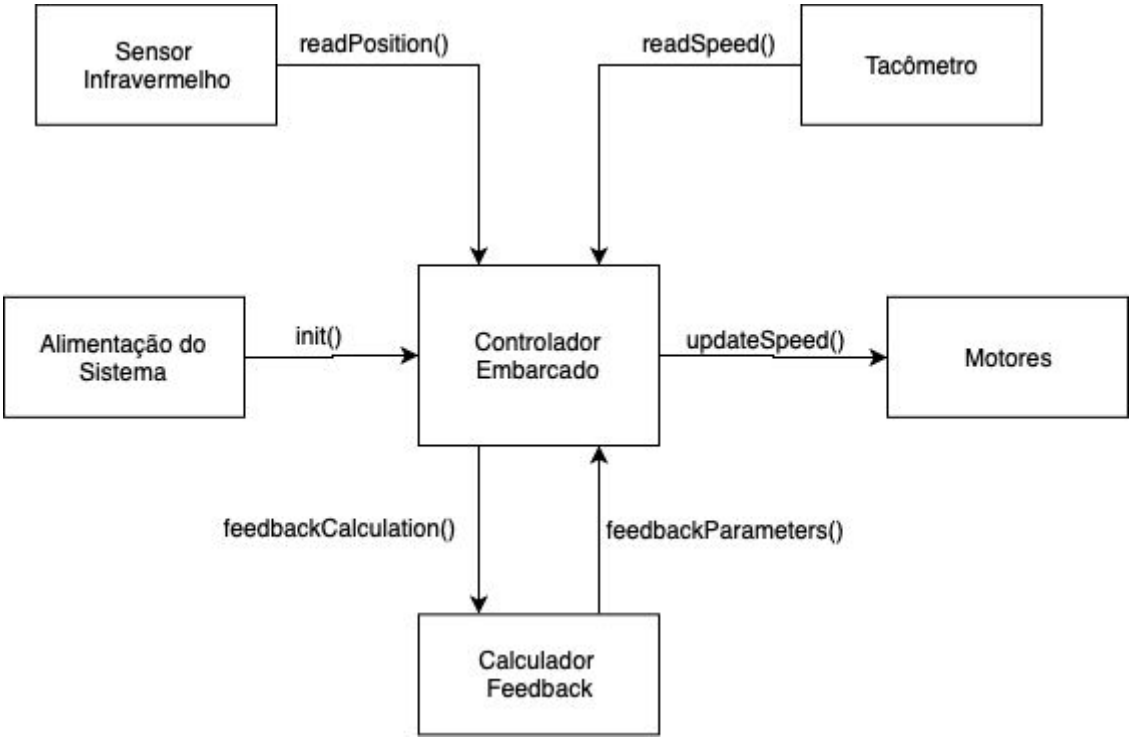


Imagem 3 - Diagrama de Comunicação

Diagrama de Classes

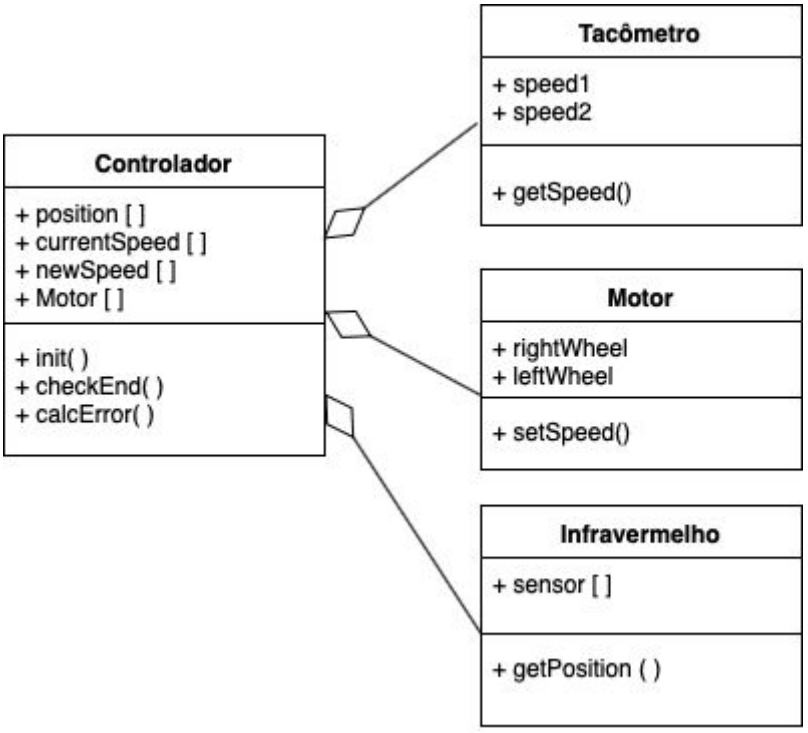


Imagem 4 - Diagrama de Classes

Diagrama de Sequência

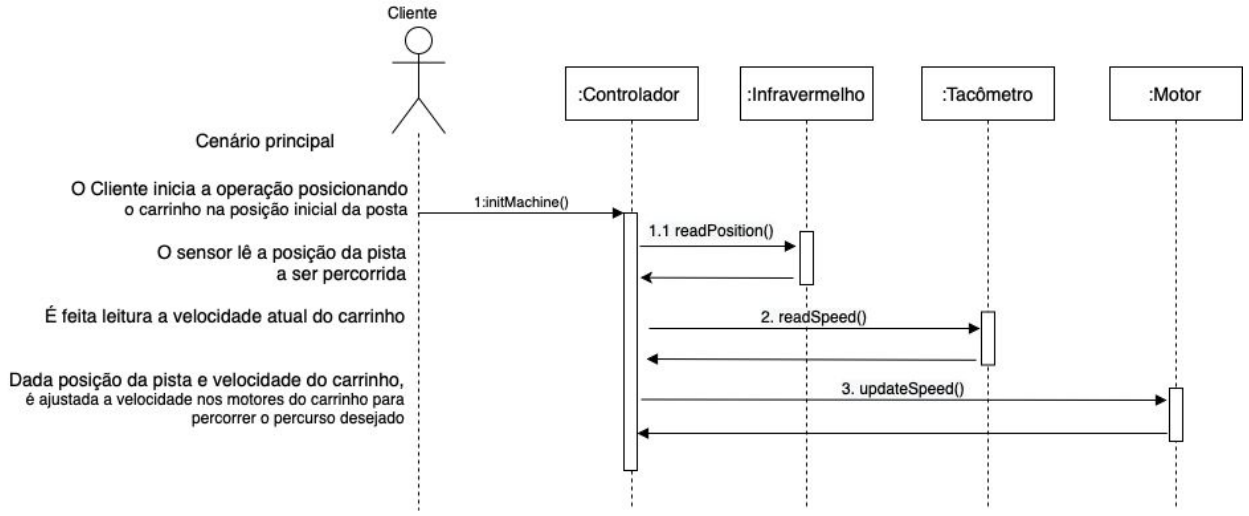


Imagem 7 - Diagrama de Sequência

Máquina de Estados

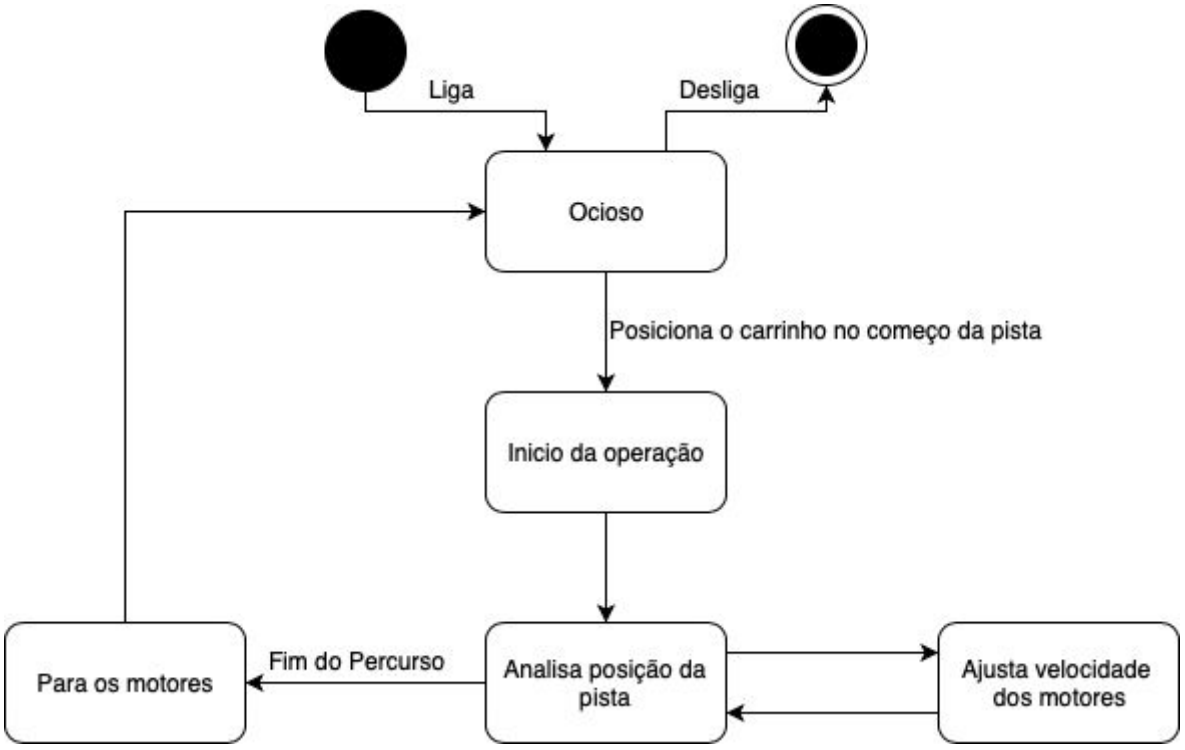


Imagem 6 - Máquina de Estados

Código Comentado

Lógica

No código, temos que controlar a velocidade das rodas do carrinho considerando a leitura dos sensores infravermelhos que indicam sua posição. Basicamente, atribuímos erros a cada posição do carrinho, ou seja, cada sensor lido possui um peso de acordo com a posição relativa na pista. Se o sensor mais a lateral esquerda estiver lendo a pista, o carrinho deve fazer uma curva à esquerda para se restabelecer no centro da pista.
Assim, os sensores externos possuem um peso maior, os internos um peso mediano e o sensor central um peso neutro. A soma desses pesos na leitura dos sensores nos fornece um sinal de erro para ser utilizado no controlador para determinar as velocidades de cada motor.
Portanto, tendo o erro da posição do carrinho, considerando os últimos erros lidos e o último ajuste feito (esforço de controle), utilizamos um controlador PID para ajustar as velocidades do carrinho, como explicado no próximo tópico.

Controlador

Considerando um controlador PID na forma abaixo

$$u(t)=K_p e(t)+K_I \times \int_{t=0}^{t} e(t)dt + K_D \times \frac{de(t)}{dt}$$

Deduzimos sua equação a diferenças para ser implementada de uma maneira mais fácil pelo microcontrolador, sem a necessidade da implementação de derivadas e integrais da fórmula acima. Com a dedução da equações a diferenças utilizando transformada Z, obtemos a seguinte equação simplificada

$$u[n] = e[n].b_0 + e[n-1].b_1 + e[n-2].b_2 - u[n-1].a_1$$

Nesta equação, consideramos o erro atual lido *e[n]*, o último erro *e[n-1]*, o penúltimo erro *e[n-2]* e o último sinal de controle calculado (esforço de controle) *u[n-1]*. Os coeficientes da equação são dados abaixo

$$b0 = Kp + Ki.h1 + Kd.h2;$$

$$b1 = -Kp + Ki.hi - 2.Kd.h2;$$

$$b2 = Kd.h2;$$

$$a1 = -1;$$

Os coeficientes *h1* e *h2* consideram o tempo de amostragem da aquisição do sinal de erro, ou seja, o tempo que o microcontrolador leva entre cada nova leitura dos sensores. No caso, o tempo foi cravado por interrupção de timer da placa ESP32. Segue, abaixo, os coeficientes

$$h_1 = \Delta t/2;$$

$$h_2 = 1/\Delta t.$$

Tendo essa equação a diferenças simplificada, conseguimos implementar no nosso microcontrolador, tomando os parâmetros *Kp*, *Kd* e *Ki* das corridas anteriores como base e ajustando os valores por experimentação.

Implementação

Com o controlador formulado e a lógica desenvolvida, implementamos o código como mostrado abaixo:

```
/* Definição de pinagem da placa para os motores */
#define engineLeftDirection1 32
#define engineLeftDirection2 33
#define engineLeftVelocity 27
#define engineRightDirection1 25
#define engineRightDirection2 26
#define engineRightVelocity 14

/* Definição de pinagem da placa para os sensores infravermelhos*/
#define sensor1 23
#define sensor2 22
#define sensor3 21
#define sensor4 19
#define sensor5 18

/* Definição das constantes para o sinal PWM */
const int freq = 30000;
const int ledChannel = 0;
const int ledChanne2 = 1;
const int resolution = 8;

/* Vetor para leitura dos sensores infravermelhos */
int LFSensor[5]={0, 0, 0, 0, 0};

/* Vetor de erros dos sensores infravermelhos */
const int trackBias[5] = {-6,-3, 0, 3, 6};
```

```
/* Velocidades dos motores */
int mean = 175;                // velocidade base
int velocidade1, velocidade2; // variaveis de velocidade para cada motor

/* Variáveis de controle para o PID */
int fimDePista = 0;            // flag para identificar final da pista
int realFim = 0;               // flag para identificar o final real do circuito
int lineOut = 0;               // flag para identificar saida da pista
bool liberaControle = false;  // flag para ativar controlador PID

/* Parâmetros do PID */
float error = 0;
float Kp = 6;
int Ki = 0;
float Kd = 3;
float deltaT = 0.1;            // período de aquisição dos valores dos sensores
float h1 = deltaT/2;           // parâmetros do PID que carregam deltaT
float h2 = 1/deltaT;
float b0, b1, b2;              // variáveis para calculo do PID
float previousErrorOne = 0;    // ultimo erro calculado [ e(n-1) ]
float previousErrorTwo = 0;    // penúltimo erro calculado [ e(n-2) ]
float PIDvalue = 0;            // resposta do controlador calculada [ u(n) ]
float PIDLastvalue = 0;        // ultima resposta do controlador calculada [
u(n-1) ]

/* Variaveis para interrupção do contador/timer */
volatile int interruptCounter;
int totalInterruptCounter;

hw_timer_t * timer = NULL;
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;

/* Interrupção do timer */
void IRAM_ATTR onTimer() {
   portENTER_CRITICAL_ISR(&timerMux);
   liberaControle = true;
   interruptCounter++;
   portEXIT_CRITICAL_ISR(&timerMux);
}
```

```
/* Definição dos outputs/inputs da pinagem da placa */
void initPinouts() {
   Serial.print("Setting pinnouts.....\n");
   pinMode(engineLeftDirection1, OUTPUT);
   pinMode(engineLeftDirection2, OUTPUT);
   pinMode(engineRightDirection1, OUTPUT);
   pinMode(engineRightDirection2, OUTPUT);

   pinMode(sensor1, INPUT);
   pinMode(sensor2, INPUT);
   pinMode(sensor3, INPUT);
   pinMode(sensor4, INPUT);
   pinMode(sensor5, INPUT);

   // Configurando funcionalidade do PWM
   ledcSetup(ledChannel, freq, resolution);
   ledcSetup(ledChanne2, freq, resolution);

   // Anexando os canais de PWM aos motores
   ledcAttachPin(engineLeftVelocity, ledChannel);
   ledcAttachPin(engineRightVelocity, ledChanne2);
   Serial.print("End setting pinnouts.....\n");
}

/* Configurando valores iniciais para a corrida */
void initRacing() {
   delay(500);
   digitalWrite(engineLeftDirection1,HIGH);
   digitalWrite(engineLeftDirection2,LOW);
   digitalWrite(engineRightDirection1,LOW);
   digitalWrite(engineRightDirection2,HIGH);
   ledcWrite(ledChannel, 160);
   ledcWrite(ledChanne2, 160);
}

/* Inicia timer por interrupção */
void initTimer() {
   timer = timerBegin(0, 80, true);
   timerAttachInterrupt(timer, &onTimer, true);
   timerAlarmWrite(timer, 50000, true);
   timerAlarmEnable(timer);
```

```
}

/* Escreve o valor de leitura dos sensores no vetor */
void readPosition() {
   LFSensor[0] = digitalRead(sensor1);
   LFSensor[1] = digitalRead(sensor2);
   LFSensor[2] = digitalRead(sensor3);
   LFSensor[3] = digitalRead(sensor4);
   LFSensor[4] = digitalRead(sensor5);

   lineOut = 0;

   // Verifica final de pista e da corrida
   if(((LFSensor[0]== 1 )&&(LFSensor[1]== 1 )&&(LFSensor[2]== 1 )&&(LFSensor[3]==
1 )&&(LFSensor[4]== 1 ))){
     if(realFim == 3){
       fimDePista += 1;
     }else{
       realFim += 1;
     }
   // Verifica saida da pista
   }else if(((LFSensor[0]== 0 )&&(LFSensor[1]== 0 )&&(LFSensor[2]== 0
)&&(LFSensor[3]== 0 )&&(LFSensor[4]== 0 ))){
       lineOut = 1;
   }else realFim = 0;
}

/* Calcula o erro do controle baseado na leitura dos sensores */
void calcError() {
   int i;
   int activeSensors = 0;
   error = 0;

   for(i=0;i<5;i++)
       activeSensors += LFSensor[i];
   // Calcula erro total dos sensores considerando os sensores ativos
   for(i=0;i<5;i++)
       error += trackBias[i]*LFSensor[i];

   // Se sair da pista, mantém o ultimo erro lido [ e(n-1) ]
   error = lineOut ? previousErrorOne : error/activeSensors;
```

```
}

/* Faz o cálculo do sinal de controle PID */
void calculatePID() {
   // Calculo das variáveis do controlador
   b0 = Kp + (Ki*h1) +(Kd*h2);
   b1 = -Kp + (Ki*h1) - (2*Kd*h2);
   b2 = Kd*h2;
   // Verifica se erro é numérico (ou se retorna NAN/null por erro de leitura dos
sensores)
   error = isnan(error) ? 0 : error;

   // Calculo do sinal de controle [ u(n) ] considerando o erro atual [ e(n) ],
ultimo erro [ e(n-1) ],
   // penúltimo erro [ e(n-2) ] e ultimo esforço de controle calculado [ u(n-1) ]
   PIDvalue = (error*b0) + (previousErrorOne*b1) + (previousErrorTwo*b2) +
PIDLastvalue;

   // Verifica se sinal de controle é numérico (ou se retorna NAN/null por erro
de cálculo)
   PIDvalue = isnan(PIDvalue) ? 0 : PIDvalue;

   // Atualiza valores antigos dos erros e esforço de controle
   previousErrorTwo = previousErrorOne;
   previousErrorOne = error;
   PIDLastvalue = PIDvalue;
}

/* Zera a velocidade dos motores para parada */
void endRace() {
   ledcWrite(ledChannel, 0);
   ledcWrite(ledChanne2, 0);
}

/* Escreve nos motores a média das velocidades +- o controle PID */
void limiteVelocidade(){
   // Limita a velocidade máxima dos motores em 220 de 255 do sinal PWM
   velocidade1 = (mean + PIDvalue) > 220 ? 220 : (mean + PIDvalue);
   velocidade2 = (mean - PIDvalue) > 220 ? 220 : (mean - PIDvalue);
}
```

```
/* Faz o setup da placa */
void setup() {
   // Inicia comunicação serial e chama métodos de configurações iniciais
   Serial.begin(115200);
   initPinouts();
   initRacing();
   initTimer();
}

/* Repete a função loop enquanto o controlador estiver ligado */
void loop() {
   unsigned long start = micros();

   // Verifica se o carrinho está no final da pista
   if(fimDePista != 2) {
       // Lê a posição do carrinho e calcula o erro
       readPosition();
       calcError();

       // Verifica flag de controle atualizada pela interrupção do contador/timer
       if(liberaControle){
           calculatePID();
           liberaControle = false;
       }
       // Atualiza velocidade dos motores
       limiteVelocidade();
       ledcWrite(ledChannel, velocidade1);
       ledcWrite(ledChanne2, velocidade2);
   }
   else {
       endRace();
   }
   delay(50);
}
```

Vídeo

O vídeo está disponível no link: https://drive.google.com/file/d/19a18mcaLapKcfCDEhLqyjargE8JJoe_Z/view?usp=sharing

Comentários

Com relação às outras corridas, esta não teve nenhuma mudança de hardware, apenas de software. A grande alteração foi no controlador utilizado. Nas primeiras corridas, não tendo uma complexidade grande no percurso, utilizamos um controlador PID simples, sem considerar o esforço de controle. Na terceira corrida, considerando esse parâmetro e aumentando o controle sobre erros anteriores, passamos a ter um controle mais robusto capaz de seguir o percurso mantendo uma boa performance.
No código, foi melhorado também o tratamento de erros de leitura, verificando sempre os valores lidos, se era realmente um valor ou se era apenas um erro de leitura (NAN ou null). Incluímos, também, uma lógica caso o carrinho saísse da pista (que não chegou a ser necessária no percurso final), em que o carrinho mantinha o último erro lido quando saísse da pista, para tentar voltar ao percurso.

Referências

THAKUR, Manoj R. Installing ESP32 Board in Arduino IDE on Ubuntu Linux. 2018. Disponível em: <https://circuits4you.com/2018/02/02/installing-esp32-board-in-arduino-ide-on-ubuntu-linux/>.

NEVES, Felipe. Controlador PID digital: Uma modelagem prática para microcontroladores - Parte 1. 2014. Disponível em: <https://www.embarcados.com.br/controlador-pid-digital-parte-1/>.

NEVES, Felipe. Controlador PID digital: Uma modelagem prática para microcontroladores - Parte 2. 2014. Disponível em: <https://www.embarcados.com.br/controlador-pid-digital-parte-2/>.

THAKUR, Manoj R. ESP32 DevKit ESP32-WROOM GPIO Pinout . 2018. Disponível em: <https://circuits4you.com/2018/12/31/esp32-devkit-esp32-wroom-gpio-pinout/>.

CANDIDO, Gradimilo. Robô seguidor de linha com sensor infravermelho e PWM. 2018. Disponível em: <https://portal.vidadesilicio.com.br/robo-seguidor-de-linha-sensor-infravermelho-e-pwm/>.

THOMSEM, Adilson. Como montar um Robô Seguidor de Linha com Arduino Motor Shield. 2014. Disponível em: <https://www.filipeflop.com/blog/projeto-robo-seguidor-de-linha-arduino/>.

THOMSEM, Adilson. Motor DC com Driver Ponte H L298N. 2013. Disponível em: <https://www.filipeflop.com/blog/motor-dc-arduino-ponte-h-l298n/>.

CARDOSO, Daniel. Ponte H L298N – Controlando a velocidade de um motor DC com PWM. 2017. Disponível em: <https://portal.vidadesilicio.com.br/ponte-h-l298n-controle-velocidade-motor/>.

THAKUR, Manoj R. ESP32 PWM Example. 2018. Disponível em: <https://circuits4you.com/2018/12/31/esp32-pwm-example/>.
SANTOS, Rui. ESP32 PWM with Arduino IDE
. 2018. Disponível em: <https://randomnerdtutorials.com/esp32-pwm-arduino-ide/>.