Guia
de
Software

Índice

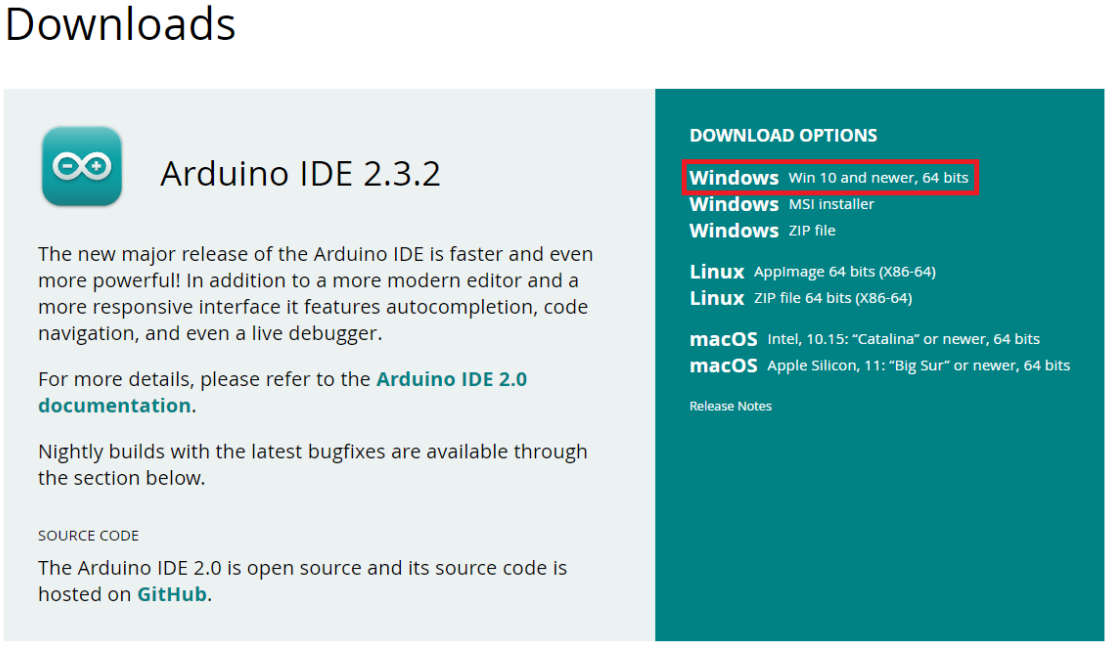
1. Instalações

1.1. Arduino IDE

A IDE (Integrated Development Environment) Arduino é um software que proporciona um ambiente de desenvolvimento completo para programar microcontroladores como Arduino e ESP32. Ela facilita o processo de escrever códigos e permite que você se concentre mais no desenvolvimento do projeto em si, sem se preocupar com detalhes técnicos complexos. Basicamente, é onde você escreve, edita, compila e carrega seus códigos (sketches) para o seu robô.

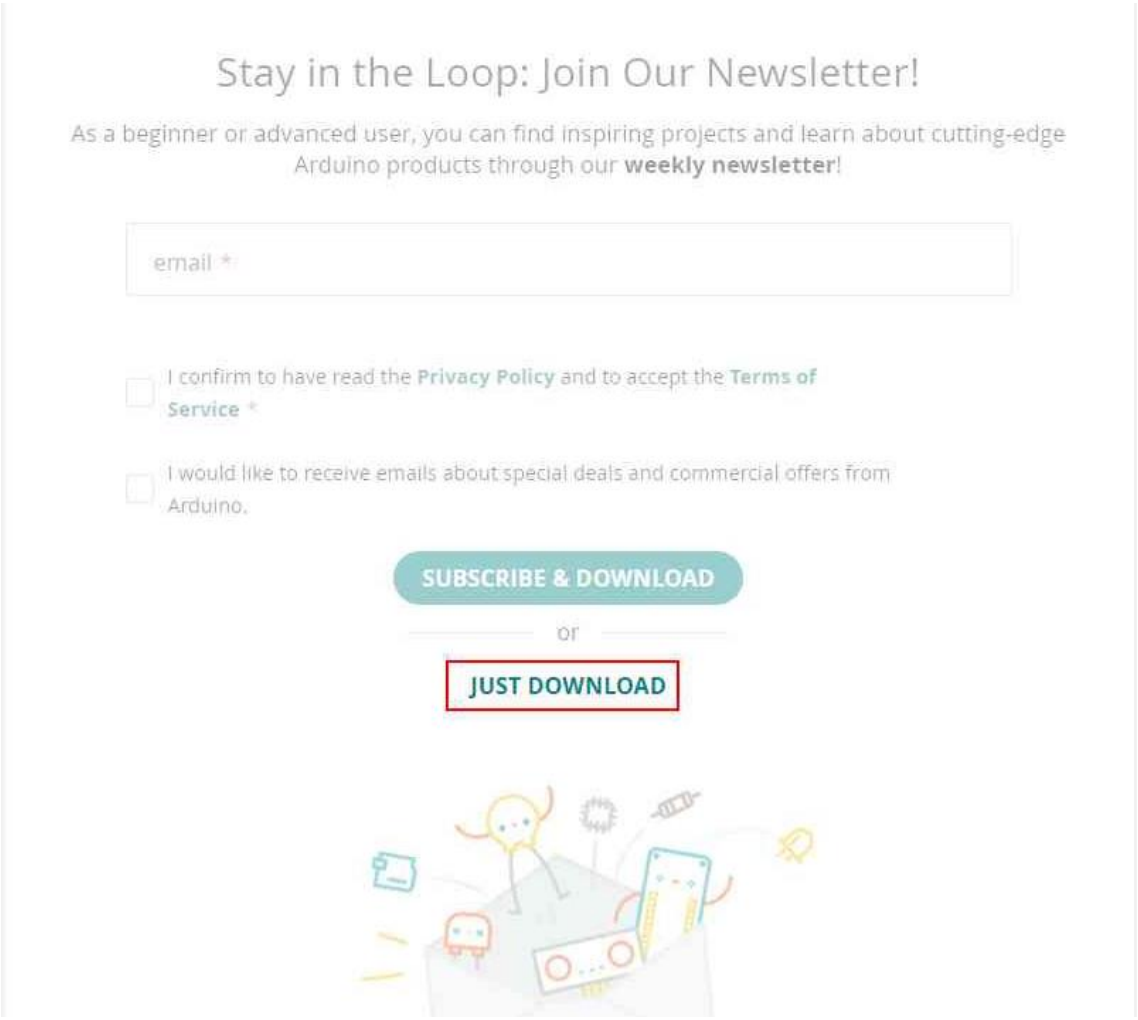
Para fazer a instalação, siga as instruções abaixo:

1º passo - Fazer o download do arquivo executável nesse link (https://www.arduino.cc/en/software).

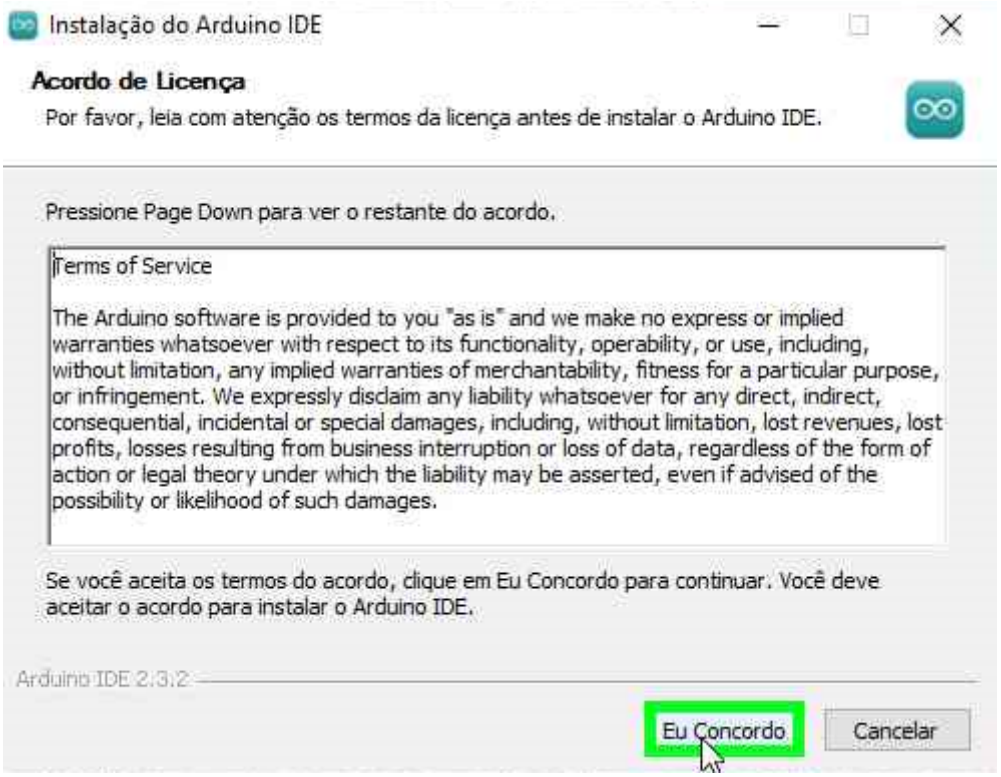


2º passo - Selecionar a opção Just Download.

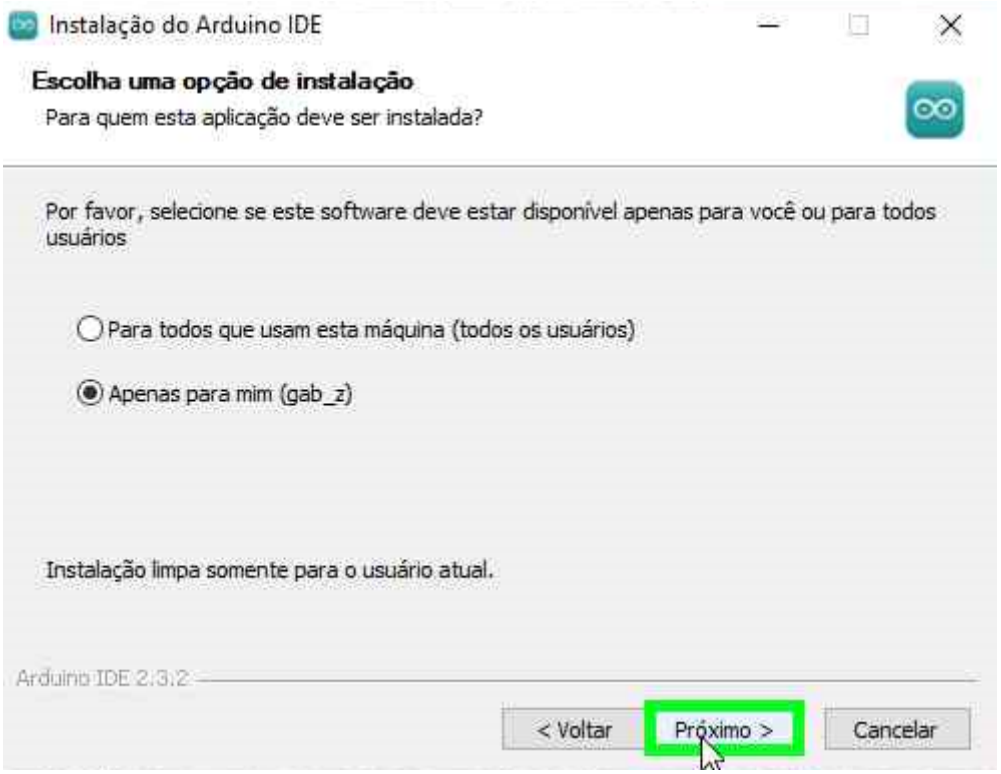
3º passo - Selecionar novamente a opção Just Download.



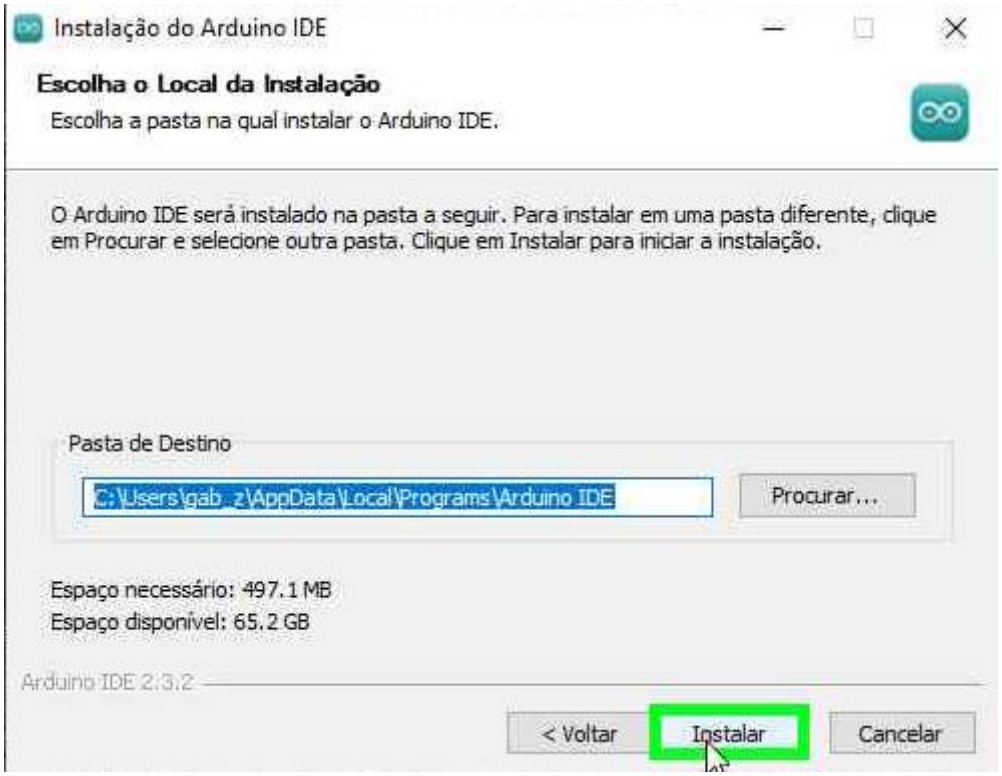
4º passo - No Instalador do Arduino IDE concorde com os termos de licença.



5º passo Em seguida escolha a opção de instalação e selecione a opção próximo.



6º passo Em seguida escolha o local da instalação e aperte a opção instalar.

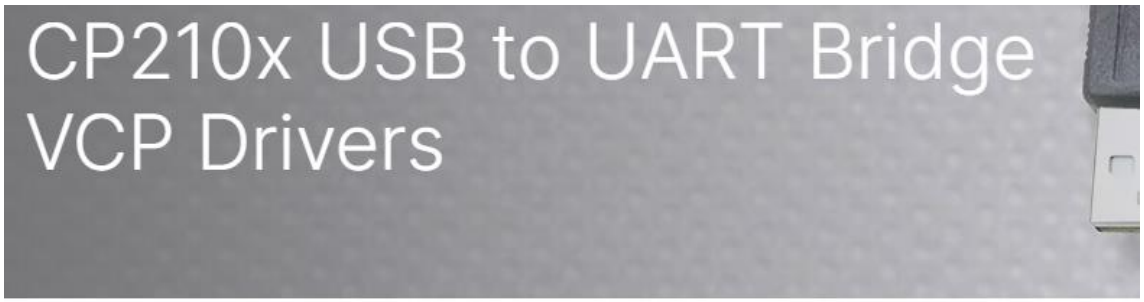


7º passo Completando a instalação aperte em concluir fazendo com que o instalador feche e execute o Arduino IDE.

1.2. Driver

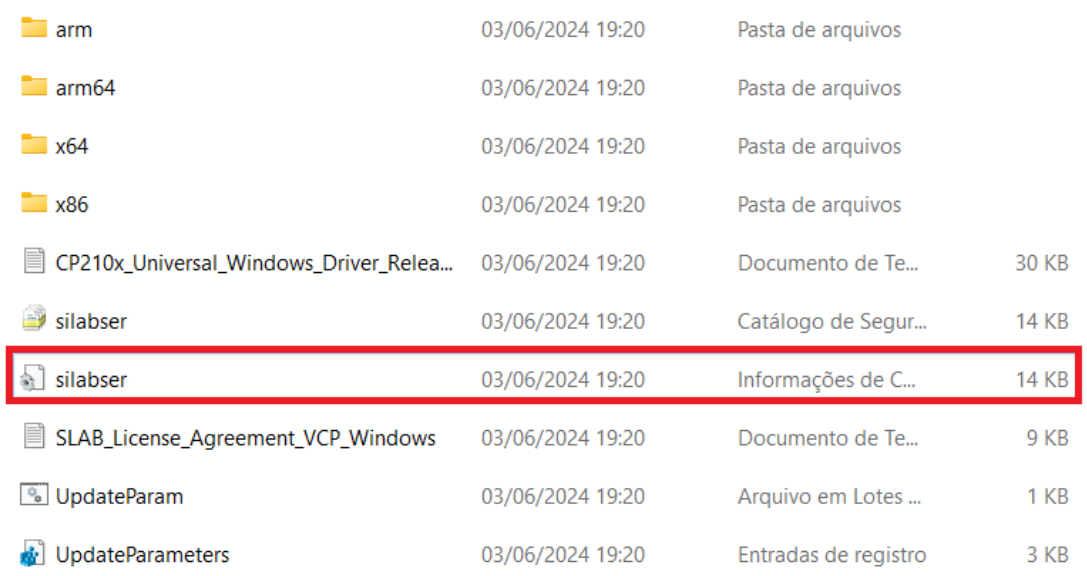
Pense que é necessário o seu computador conversar com a ESP para que seja possível enviar os códigos para a ela. Porém, caso você apenas conecte a ESP ao seu computador, ele não será capaz de determinar qual dos dispositivos conectados (mouse, teclado, fone de ouvido, monitor) é o dispositvo em questao. É exatamente nesse momento que entra o driver, para que ele possa especificar para o computador qual dispositivo é o correto para a comunicação de dados. Caso queira saber mais acesse o link: https://learn.microsoft.com/pt-br/windows-hardware/drivers/gettingstarted/what-is-a-driver-

1º passo - Fazer o download do arquivo executável nesse link (https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers?tab=overview).

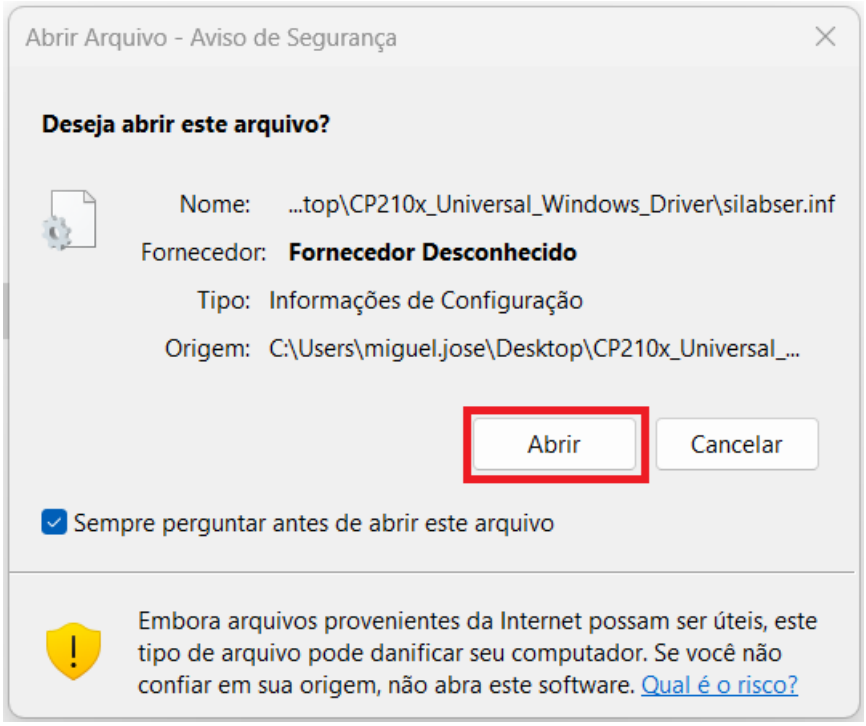


2º passo – Selecione a primeira opção de Download.

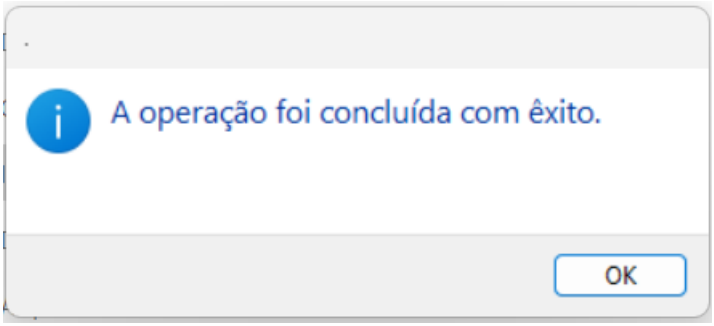
3º passo – Ao baixar abra a pasta que você realizou o download e selecione o arquivo silabser.



4º passo – Ao abrir o arquivo recebera uma confirmação se deseja abrir precisione abrir normalmente.



5º passo – Apos abrir o driver será instalado assim que a instalação acabar você recebera uma mensagem de confirmação que a instalação esta concluída com êxito basta aperta a opção ok.



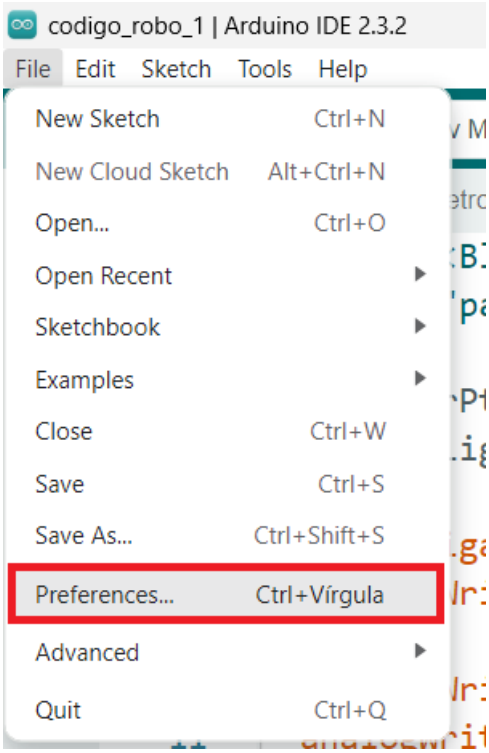
1.3. Bibliotecas

Pense na biblioteca como sendo um livro de referência que o código vai usar para saber como realizar certas funções. A IDE possui um gerenciador de bibliotecas que facilita a adição das mesmas para expandir as capacidades do microcontrolador, como controlar sensores, módulos Wi-Fi, displays, etc. No nosso caso, vamos usar uma biblioteca para auxiliar no uso do controle por bluetooth.

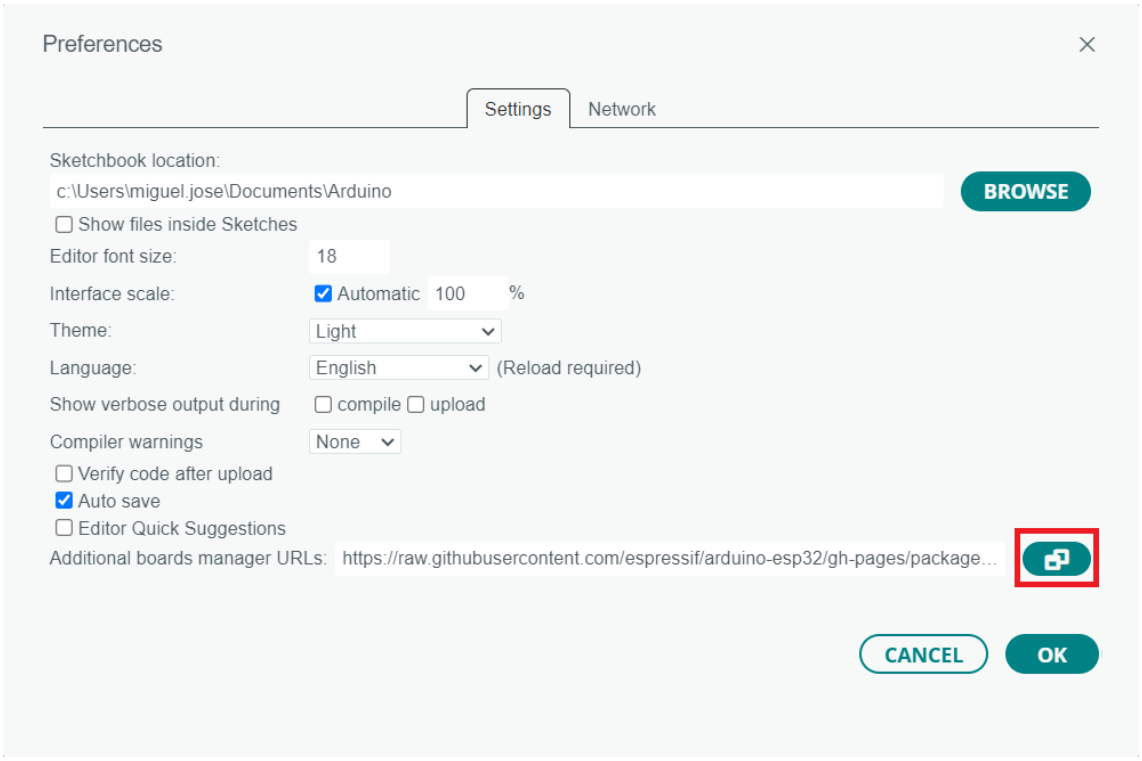
No menu Sketch, você pode selecionar e incluir no seu projeto as bibliotecas desejadas para realizar funções específicas.

No nosso projeto utilizaremos a biblioteca Bluepad32, a documentação completa da mesma pode ser acessada nesse link (https://bluepad32.readthedocs.io/en/latest/).

1º passo – Abra o Arduino IDE vá em File selecione a opção Preferences (ou utilize o atalho Ctrl + Vírgula).



2º passo – Já na aba Preferences selecione a opção Settings e selecione a opção com dois quadradinho como a imagem abaixo demonstra.

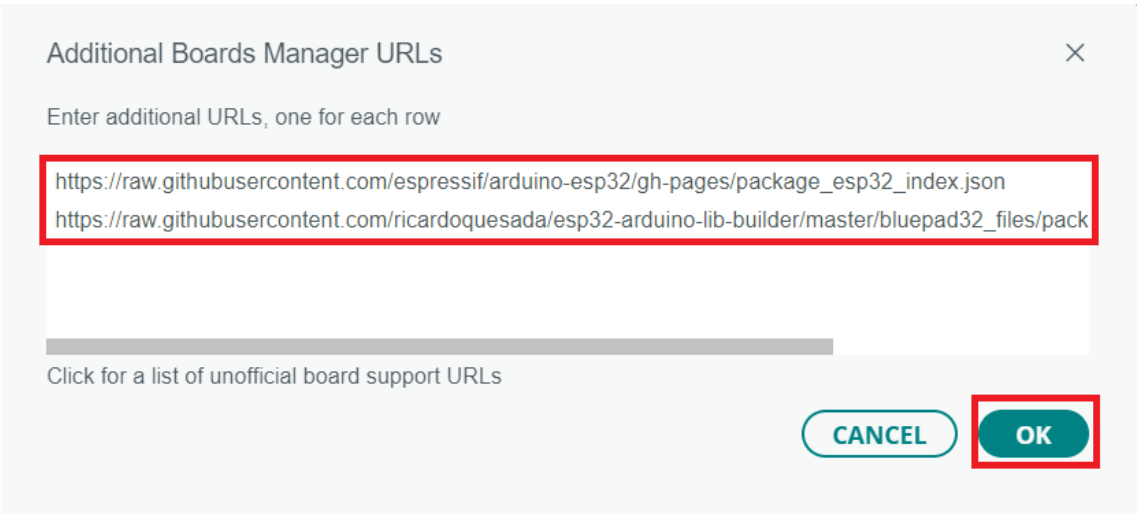


3º passo – Abrindo essa opção você terá o Additional Boards Manager URLs onde você ira colar estes dois links abaixo.

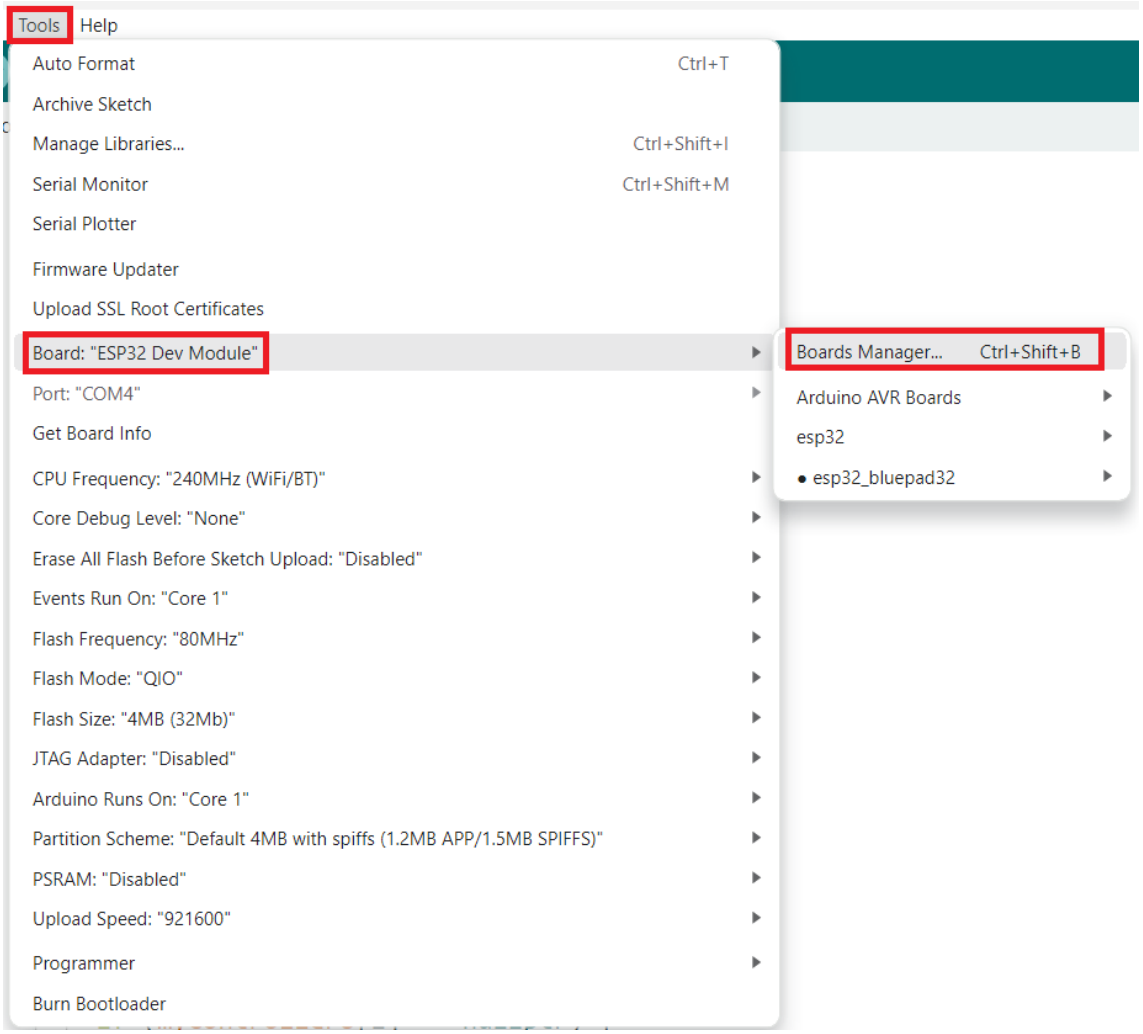
Link 1: https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json

Link 2: https://raw.githubusercontent.com/ricardoquesada/esp32-arduino-lib-builder/master/bluepad32_files/package_esp32_bluepad32_index.json

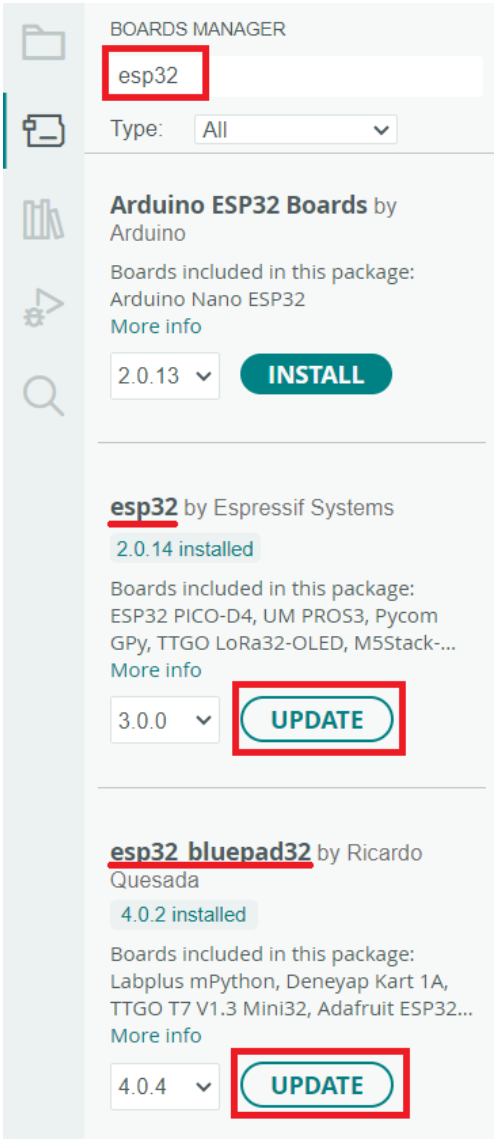
Em seguida pressione a opção Ok.



4º passo – Voltando para o Arduino IDE vá em Tools selecione a opção Board -> Boards Manager... (ou pressione Ctrl+Shift+B).



5º passo – Em BOARDS MANAGER digite esp32 e realize o install ou o update das bibliotecas esp32 e esp32 bluepad32.



2. Linguagem Arduino

A linguagem de programação do Arduino é baseada em C/C++ e será a linguagem que utilizaremos para programar nosso microcontrolador. Ela pode ser dividida em três partes principais: estruturas, valores (variáveis e constantes) e funções.

Abaixo apresentaremos um resumo de cada uma dessas partes, mas caso queira se aprofundar mais no assunto, aconselhamos que consulte a documentação completa do Arduino nesse link (https://www.arduino.cc/reference/pt/).

2.1. Valores

Os valores são tipos de dados e constantes da linguagem Arduino, vamos ver alguns deles a seguir.

2.1.1. Constantes

Constantes em Arduino são valores que permanecem inalterados durante a execução do programa. Elas são usadas para dar nomes significativos a valores fixos,

tornando o código mais legível e fácil de manter. Existem várias maneiras de definir constantes em Arduino, veremos algumas abaixo:

2.1.1.1. Constantes Booleanas

Existem duas constantes usadas para representar verdade e falsidade na linguagem Arduino:

- **true**

Alegação de verdade. Normalmente definido como 1, também pode ter uma definição mais ampla. Qualquer inteiro que não seja zero é considerado *true*, em um sentido booleano. Então -1, 2 e -200 são todos definidos como *true*, também, em um sentido booleano.

- **false**

Alegação de falsidade, é o mais fácil de ser definido. Sendo apenas considerado como *false* o inteiro 0 (zero).

2.1.1.2. Modos para Pinos Digitais

Em Arduino, definir os modos dos pinos digitais é fundamental para controlar como eles interagem com o ambiente externo. A definição de modos é feita com a função pinMode(), que configura um pino específico para operar como entrada (INPUT) ou saída (OUTPUT). Essa configuração será feita dentro do bloco de código *setup()*, explicado mais à frente. Vamos entender um pouco mais sobre esses modos:

- **Entrada (INPUT)**

Podemos considerar como entrada todo sinal que será enviado para o microcontrolador partindo do meio externo, como botões ou sensores. Exemplo de configuração de um pino em INPUT:

```
pinMode(2, INPUT);
```

O trecho de código acima configura o pino 2 do microcontrolador como sendo de entrada.

- **Saída (OUTPUT)**

Podemos considerar como saída todo sinal que será enviado do microcontrolador para o meio externo, como acionar LEDs ou ligar motores.
Exemplo de configuração de um pino em OUTPUT:

```
pinMode(13, OUTPUT);
```

O trecho de código acima configura o pino 13 do microcontrolador como sendo de saída.

2.1.1.3. Níveis Lógicos

Os níveis lógicos são usados para definir os estados dos pinos digitais, tanto para leitura quanto para escrita. Há apenas dois valores possíveis que esses pinos podem assumir:

- **HIGH**

Representa nível lógico alto, em estado binário é igual a 1. Isso significa que, no caso de leitura, o retorno seria HIGH se o pino estivesse ligado. Em caso de escrita, você estaria estabelecendo o estado do pino como ligado.

- **LOW**

Representa nível lógico baixo, em estado binário é igual a 0. Isso significa que, no caso de leitura, o retorno seria LOW se o pino estivesse desligado. Em caso de escrita você estaria estabelecendo o estado do pino como desligado.

2.2. Estruturas

As estruturas são os elementos da linguagem Arduino, vamos ver algumas delas a seguir.

2.2.1. Sketch

Um esboço é o nome que o Arduino usa para um programa. É a unidade de código que é carregada e executada em uma placa Arduino. Um sketch tem a seguinte estrutura:

```
1  void setup() {
2    // put your setup code here, to run once:
3
4  }
5
6  void loop() {
7    // put your main code here, to run repeatedly:
8
9  }
10
```

Como você pode ver, ele é dividido em dois grandes blocos: setup e loop. Vamos entender um pouquinho mais sobre cada um deles.

A função *setup()* é chamada quando um sketch inicia. Ela pode ser utilizada para inicializar variáveis, configurar o modo dos pinos (INPUT ou OUTPUT), inicializar bibliotecas, etc. Essa função será executada apenas uma vez, após a placa ser alimentada ou acontecer um reset.

A função *loop()* faz precisamente o que o seu nome sugere, e repete-se consecutivamente enquanto a placa estiver ligada, permitindo o seu programa mudar e responder a essas mudanças. Use-a para controlar ativamente uma placa Arduino.

Sendo assim, podemos dizer que na função *setup()* colocaremos todas as configurações estáticas do nosso programa e na função *loop()* colocaremos toda a lógica do nosso programa.

2.2.2. Elementos de Sintaxe

Quando programando em linguagem de Arduino, é muito importante se manter atento aos elementos de sintaxe. Eles permitem que seu código fique mais dinâmico e legível. Vamos ver alguns deles abaixo:

- **#define**

É uma diretiva muito útil que te permite dar um nome a um valor constante antes de o programa ser compilado. Assim, sempre que você for usar aquele valor, poderá chamá-lo pelo seu nome. É importante ressaltar que a declaração de #defines deve ser feita antes, fora dos blocos *setup()* e *loop()*.

```
#define LED 3
```

No exemplo acima, definimos que no pino 3 do microcontrolador, se encontra conectado um LED. Portanto, toda vez que formos usar o LED no código, poderemos chamá-lo apenas de LED, sem precisar procurar em qual pino ele está conectado.

- **#include**

Essa diretiva é usada para incluir bibliotecas (veremos mais sobre esse conceito à frente) externas ao seu sketch. Isso dá acesso a um grande número de bibliotecas padrão da linguagem C (grupos de funções prontas), e também bibliotecas escritas especialmente para a linguagem Arduino.

```
#include <Bluepad.h>
```

No exemplo acima, estamos adicionando a biblioteca Bluepad no código, essa será a biblioteca que usaremos para controlar nosso robô.

- **Comentários**

Comentários são textos no programa normalmente usados para descrever a forma como o programa funciona, eles são ignorados pelo compilador e por isso não atrapalham o funcionamento do código. É uma boa prática fazer sempre comentários que te auxiliem a lembrar o motivo pelo qual o código foi escrito daquela maneira. Os comentários podem ser escritos de duas maneiras:

```
// Esse é um comentário de uma linha
/* Esse é um comentário de bloco
posso continuar escrevendo
em outras linhas
até que o bloco seja fechado
com esse sinal*/
```

- **Ponto e vírgula**

Usado para encerrar um comando. É muito importante ressaltar que o ponto e vírgula **não pode** ser esquecido! A ausência do ponto e vírgula no final de um comando resulta em erro de compilação. Isso significa que o microcontrolador não vai entender o código e por isso ele não irá executar.

- **Chaves**

As chaves são usadas em diversas estruturas diferentes, elas delimitam um bloco de códigos que será executado dentro de certas condições. Uma chave "{" deve ser sempre fechada por outra chave "}". Essa é uma condição que é frequentemente chamada de balanceamento das chaves. Assim como o ponto e

vírgula, toda chave aberta **tem que ser fechada**! Caso contrário, seu código pode apresentar erros enigmáticos e muitas vezes difíceis de serem localizados.

2.2.3. Estruturas de Controle

Estruturas de controle são comandos que alteram o fluxo de execução do programa, permitindo tomar decisões, repetir blocos de código e controlar o comportamento do programa de maneira dinâmica. Veremos abaixo alguns exemplos:

- **if**

O comando *if* checa uma condição e, se a condição é verdadeira, executa o comando a seguir ou um bloco de comandos delimitados por chaves.

```
if (condição) {
  comando(s);
}
```

- **else**

A combinação *if… else* permite maior controle sobre o fluxo de código que o comando mais básico *if*, por permitir o agrupamento de múltiplos testes. Uma cláusula *else* será executada se a condição do comando *if* é falsa.

```
if (condição) {
  comando(s);
}else{
  comando(s);
}
```

- **else if**

O *else* pode proceder outro teste *if*, formando então um novo bloco denominado *else if*. Note que um bloco *else if* pode ser usado sem um bloco *else* no final e vice-versa. Um número praticamente ilimitado de blocos *else if* conectados é permitido.

```
if (condição) {
  comando(s);
}else if (condição2){
  comando(s);
}
```

- **for**

O comando *for* é usado para repetir um bloco de código envolvido por chaves. Um contador de incremento é geralmente utilizado para terminar o loop criado pelo *for*. O comando *for* é útil para qualquer operação repetitiva, e é usado frequentemente com vetores para operar em coleções de dados ou pinos.

```
for (inicialização; condição; incremento) {
  comando(s);
}
```

2.2.4. Operadores Aritméticos

Os operadores Aritméticos são fundamentais para realizar cálculos matemáticos básicos. Esses operadores são utilizados para manipular variáveis e constantes. Mas é importante ressaltar que **a estrutura aqui é diferente da que estamos acostumados!** Veremos abaixo alguns exemplos:

- **Adição (+)**

O operador de *adição* é usado para somar dois valores.

```
resultado = valor1 + valor2;
```

Então para realizar a soma:

2 + 2 = 4

Na realidade faríamos:

```
resultado = 2 + 2;
```

E o resultado seria igual a 4.

- **Subtração (-)**

O operador de *subtração* é usado para subtrair dois valores.

```
resultado = valor1 - valor2;
```

- **Multiplicação (*)**

O operador de multiplicação é usado para multiplicar dois valores.

```
resultado = valor1 * valor2;
```

- **Divisão (/)**

O operador de divisão é usado para dividir dois valores. É importante ressaltar que, ao dividir inteiros, o resultado será um número inteiro (a parte fracionária é descartada). Para obter resultados de divisão com ponto flutuante, é necessário usar variáveis do tipo *float* ou *double.*

```
resto = divisor / dividendo;
```

- **Resto (%)**

O operador de resto é usado para obter o restante da divisão de um valor por outro.

```
resto = divisor % dividendo;
```

- **Atribuição (=)**

O sinal de igual = na linguagem de programação Arduino é chamado operador de atribuição. Seu significado é diferente das aulas de matemática, onde ele indica uma equação ou igualdade. O operador de atribuição indica ao microcontrolador que compute qualquer valor ou expressão que estiver à direita do sinal de igual, e o armazene na variável à esquerda do sinal de igual.

```
int valor;
valor = 1;
```

2.2.5. Operadores De Comparação

Os operadores de comparação são fundamentais, eles permitem a avaliação de expressões e são essenciais para estruturas de controle como o *if* e o *for*, vistos anteriormente. Veremos abaixo alguns exemplos:

- **Igual a (==)**

O operador "Igual a" compara a variável à esquerda com o valor ou variável à direita do operador. Retorna verdadeiro (*true*) quando os dois operandos são iguais.

```
x == y;
```

Exemplo de utilização com a estrutura de controle *if*:

```
if (x == y) {
  comando(s);
}
```

- **Diferente de (!=)**

O operador "Diferente de" compara a variável à esquerda com o valor ou variável à direita do operador. Retorna verdadeiro (*true*) quando os operandos não são iguais.

```
x != y;
```

Exemplo de utilização com a estrutura de controle *for*:

```
for(x=0, x!=y, x++){
  comando(s);
}
```

- **Menor que (<)**

O operador "Menor que" compara a variável à esquerda com o valor ou variável à direita do operador. Retorna verdadeiro (*true*) quando o operando à esquerda é menor que o operando à direita.

```
x < y;
```

- **Maior que (>)**

O operador "Maior que" compara a variável à esquerda com o valor ou variável à direita do operador. Retorna verdadeiro (*true*) quando o operando à esquerda é maior que o operando à direita.

```
x > y;
```

- **Menor que ou igual a (<=)**

O operador "Menor que ou igual a" compara a variável à esquerda com o valor ou variável à direita do operador. Retorna verdadeiro (*true*) quando o operando à esquerda é menor ou igual ao operando à direita.

```
x <= y;
```

- **Maior que ou igual a (>=)**

O operador "Maior que ou igual a" compara a variável à esquerda com o valor ou variável à direita do operador. Retorna verdadeiro (*true*) quando o operando à esquerda é maior ou igual ao operando à direita.

```
x >= y;
```

2.2.6. Operadores Booleanos

Os operadores booleanos são essenciais para escrever expressões condicionais e controlar o fluxo do programa. Assim como os operadores de comparação, os operadores booleanos também são essenciais para estruturas de controle como o *if* e o *for*. Veremos abaixo alguns exemplos:

- **NÃO lógico (!)**

O operador "NÃO lógico" resulta em verdadeiro se o operando é falso, e vice-versa. Ele inverte o valor lógico de uma expressão.

```
x = !y;
```

No exemplo acima, o valor lógico invertido de y é armazenado em x. Suponhamos que y = *true*, então o valor armazenado em x seria *false*.

- **E lógico (&&)**

O operador "E lógico" resulta em verdadeiro, **apenas se ambos** operandos são verdadeiros. Caso contrário, ele resulta em falso.

Exemplo utilizando a estrutura de controle *if*:

```
if (condição1 && condição2) {
  comando(s);
}
```

No exemplo acima, os comandos seriam executados apenas se ambas condições fossem verdadeiras.

- **OU lógico (!)**

O operador "OU lógico" resulta em verdadeiro se **pelo menos um** dos operandos é verdadeiro.

Exemplo utilizando a estrutura de controle *for*:

```
for (x = 0, condição1 || condição2, x++) {
  comando(s);
}
```

No exemplo acima, os comandos seriam executados enquanto pelo menos uma das condições fosse igual.

2.3. Funções

Funções são blocos de código que executam tarefas específicas. Existem diversas funções nativas da linguagem, feitas para auxiliar o processo de criação de um código. Vamos ver algumas delas a seguir:

2.3.1. Leitura e Escrita de Dados

Já vimos anteriormente o conceito de entrada e saída de dados (INPUT e OUTPUT). Agora precisamos definir como o microcontrolador vai, de fato, utilizar os pinos declarados como sendo de entrada e saída. Para isso, temos algumas funções prontas que realizam a leitura e escrita de dados. Eles podem ser de dois tipos: digitais e analógicos. Vamos entender melhor o funcionamento dessas funções e quando usá-las a seguir.

2.3.1.1. Leitura e Escrita Digitais

Um sinal digital é um tipo de sinal que alterna entre dois níveis distintos de tensão, representando os valores binários 0 e 1.

- **Modo (pinMode)**

Configura o pino especificado para funcionar como uma entrada ou saída. Essa configuração deve ser feita dentro da função *setup()*.

```
pinMode(pino, modo);
```

- **Leitura (digitalRead)**

Lê o valor de um pino digital especificado, e retorna HIGH ou LOW.

```
digitalRead(LED);
```

No exemplo acima, é realizada a leitura do estado do pino LED. O retorno será HIGH (caso esteja ligado) ou LOW (caso esteja desligado).

- **Escrita (digitalWrite)**

Aciona um valor HIGH ou LOW em um pino digital.

```
digitalWrite(LED, HIGH);
```

No exemplo acima, o pino LED recebe o valor HIGH, com isso ele é ligado. Para desligá-lo, bastaria trocar HIGH por LOW.

2.3.1.2. Leitura e Escrita Analógicas

Um sinal analógico é um tipo de sinal que varia continuamente ao longo do tempo, podendo assumir qualquer valor dentro de um intervalo específico (no nosso caso, de 0 a 5V).

- **Leitura (analogRead)**

Lê o valor de um pino digital especificado, e retorna a leitura analógica (int).

```
analogRead(LED);
```

No exemplo acima, é realizada a leitura do estado do pino LED. O retorno será um número inteiro que indica o brilho do LED.

- **Escrita (analogWrite)**

Aciona um valor (int) em um pino analógico.

```
analogWrite(LED, 255);
```

No exemplo acima, o pino LED recebe o valor 255, isso significa que ele receberá o valor máximo de brilho. Para desligá-lo, bastaria trocar 255 por 0. Valores intermediários também são aceitos para configurar outros níveis de brilho.

3. Código do robô

3.1. Importando bibliotecas

Nas duas primeiras linhas do programa temos a inclusão da biblioteca utilizada para fazer a comunicação com o controle de PS3 (Bluepad32) e a inclusão do arquivo de configurações. O arquivo de configurações será explicado posteriormente.

```
#include <Bluepad32.h>
#include "parametros.h"
```

3.2. Definição de variáveis

Logo após as inclusões temos a definição de duas variáveis que serão utilizadas ao longo do programa. A variável myControllers representa o controle que iremos conectar à ESP32. Já a variável roboLigado é um indicador do estado do robô: ligado (verdadeiro/true) ou desligado (falso/false).

```
ControllerPtr myControllers[BP32_MAX_GAMEPADS];
bool roboLigado;
```

3.3. Funções

Uma função em Arduino é um bloco de código que executa uma tarefa específica e pode ser chamado repetidamente ao longo do programa. As funções ajudam a organizar e reutilizar o código de forma eficiente.

3.3.1. desligaRobo()

A função desligaRobo() contém todos os comandos necessários para interromper o funcionamento do robô. Nas primeiras linhas da função utilizamos o comando digitalWrite para desligar (LOW) as saídas associadas ao controle da arma e ao controle dos motores de movimentação. Em sequência, definimos a velocidade de rotação dos motores de locomoção como sendo 0, utilizando o comando analogWrite. Ao final, definimos o estado do robô como desligado (roboLigado = false).

```
void desligaRobo() {
  digitalWrite(PINO_ARMA, LOW);

  digitalWrite(SENTIDO_MOTOR_ESQUERDO, LOW);
  analogWrite(VELOCIDADE_MOTOR_ESQUERDO, 0);

  digitalWrite(SENTIDO_MOTOR_DIREITO, LOW);
  analogWrite(VELOCIDADE_MOTOR_DIREITO, 0);

  roboLigado = false;
}
```

3.3.2. onConnectedController()

A função onConnectedController() é a responsável por conectar o controle à ESP32. Caso deseje realizar alguma tarefa assim que o controle for conectado, basta adicionar o trecho de código no local indicado (entre o comentário e o comando break). É importante ressaltar que a execução dessa função é feita de forma automática pela ESP32, no momento em que há uma conexão/desconexão do controle.

```
void onConnectedController(ControllerPtr ctl) {

  bool foundEmptySlot = false;

  for (int i = 0; i < BP32_MAX_GAMEPADS; i++) {
    if (myControllers[i] == nullptr) {
      Serial.println("AVISO: controle conectado.");
      myControllers[i] = ctl;
      foundEmptySlot = true;

      /* Caso deseje realizar alguma tarefa assim que a conexão
         com o contole for estabelecidada, coloque o código aqui. */

      break;
    }
  }

  if (!foundEmptySlot) {
    Serial.println("AVISO: Nao foi possivel conectar o controle.");
    Serial.println("AVISO: Reinicie a ESP32 e tente novamente.");
  }
}
```

3.3.3. onDisconnectedController()

A função onDisconnectedController() é a responsável por desconectar o controle da ESP32. Caso deseje realizar alguma tarefa assim que o controle for desconectado, basta adicionar o trecho de código no local indicado (entre o comentário e o comando break). É importante ressaltar que a execução dessa função é feita de forma automática pela ESP32, no momento em que há uma conexão/desconexão do controle.

```
void onDisconnectedController(ControllerPtr ctl) {
  for (int i = 0; i < BP32_MAX_GAMEPADS; i++) {
    if (myControllers[i] == ctl) {
      Serial.printf("AVISO: controle desconectado");
      myControllers[i] = nullptr;
      desligaRobo();

      /* Caso deseje realizar alguma tarefa assim que o
         controle for desconectado, coloque o código aqui */

      break;
    }
  }
}
```

Tanto a função onConnectedController() quanto a função onDisconnectedController() são definidas pela própria biblioteca que estamos utilizando para fazer a interface com o controle (Bluepad32). Entretanto, algumas modificações foram feitas a fim de deixá-las mais simples e compactas.

Uma chamada da função desligaRobo() foi adicionada na função onDisconnectedController() para garantir que o robô pare sua movimentação e sua arma em caso de perda de conexão com o controle (sistema de proteção exigido nas regras de competição da categoria cupim).

3.3.4. processControllers()

A função processControllers() é a responsável por mapear o que foi pressionado no controle e, a partir dessas entradas, determinar o funcionamento do robô. Portanto, **toda a lógica de funcionamento do robô está descrita dentro desta função.**

```
void processControllers() {
  for (auto myController : myControllers) {

    if (myController && myController->isConnected()
        && myController->hasData() && myController->isGamepad()) {

      /* A partir daqui inicia-se a lógica de funcionamento do robô.
         Qualquer alteração / nova implementação deve ser feita aqui. */
```

A primeira tarefa realizada pela função é confirmar se há um dispositivo *bluetooth* conectado. Logo após, ela também verifica se o dispositivo está enviando dados e se ele é, de fato, um controle (Gamepad). Essa última verificação é feita pois a mesma biblioteca pode ser utilizada para a conexão com diversos tipos de dispositivos *bluetooth* (mouse, teclado, ...). Se todas essas condições forem satisfeitas, a leitura e o processamento dos dados do controle são feitos.

Feita a verificação da conexão, lê-se o estado dos botões SELECT e START. Se SELECT for pressionado (estadoMiscButtons == 0x02), desliga-se o robô (roboLigado =

false). Caso contrário, se START for pressionado (estadoMiscButtons == 0x04), liga-se o robô (roboLigado = true).

```
uint16_t estadoMiscButtons = myController->miscButtons();

// Se SELECT for presionado, desliga robô
if (estadoMiscButtons == 0x02) {
  roboLigado = false;
  Serial.println("Robo desligado");
}

// Se START for presionado, liga robô
else if (estadoMiscButtons == 0x04) {
  roboLigado = true;
  Serial.println("Robo iniciado");
}
```

Se o robô estiver ligado, lê-se o estado dos botões (X, O, L1, R1, ...). Se L1 for pressionado (botoesPressionados == 0x0010), desliga-se a arma (LOW). Senão, se R1 for pressionado (botoesPressionados == 0x0020), liga-se a arma (HIGH).

```
uint16_t botoesPressionados = myController->buttons();

// Se L1 for presionado, desliga arma.
if (botoesPressionados == 0x0010) {
  digitalWrite(PINO_ARMA, LOW);
  Serial.print("Arma desligada ");
}

// Se R1 for presionado, liga arma.
else if (botoesPressionados == 0x0020) {
  digitalWrite(PINO_ARMA, HIGH);
  Serial.print("Arma ligada ");
}
```

Ainda se o robô estiver ligado, o valor vertical do analógico direito (valorAnalogicoDireito) e o valor horizontal do analógico esquerdo (valorAnalogicoEsquerdo) são lidos e exibidos (Serial.print()) no Monitor Serial.

De forma geral, o analógico retorna um valor negativo se pressionado para a esquerda e um valor positivo se pressionado para a direita. Já na vertical, o analógico retorna um valor negativo se pressionado para cima e um valor positivo se pressionado

para baixo. De acordo com a documentação da Bluepad32, para ambas direções, horizontal e vertical, o valor retornado varia de -511 a 512.

```
// Lê valor em Y do analógico direito (R-right)
int32_t valorAnalogicoDireito = -myController->axisRY();

// Lê valor em X do analógico esquerdo (L-left)
int32_t valorAnalogicoEsquerdo = myController->axisX();

// Exibe valores no monitor serial

Serial.print("Y analogico R: ");
Serial.println(valorAnalogicoDireito);

Serial.print("X analogico L: ");
Serial.println(valorAnalogicoEsquerdo);
```

O valor vertical do analógico direito está sendo multiplicado por -1 para deixar no mesmo sentido do plano cartesiano (cima +, baixo -), facilitando a compreensão. Considere esta informação ao longo das explicações abaixo.

3.4. Movimentação

Feita a leitura dos joysticks, o programa determina a velocidade e o sentido de rotação de cada motor baseado no valor lido. A lógica de determinação é bastante simples:

3.4.1. Frente/Direita

Se o analógico direito for pressionado para frente (+) e o analógico esquerdo for pressionado para a direita (+), o robô deve andar para frente e para direita. Para esse caso, a sentido de rotação de ambas rodas deve ser igual. Entretanto, a velocidade da roda esquerda deve ser maior que a velocidade da roda direita, para que o robô vire para a direita.

```
int pwmMotorDireito, pwmMotorEsquerdo;

if (valorAnalogicoDireito > (centerAnalogR_Y + toleranciaAnalogico)) {

  digitalWrite(SENTIDO_MOTOR_DIREITO, HIGH);
  digitalWrite(SENTIDO_MOTOR_ESQUERDO, HIGH);

  if (valorAnalogicoEsquerdo > (centerAnalogL_X + toleranciaAnalogico)) {

    pwmMotorDireito = map(valorAnalogicoDireito - valorAnalogicoEsquerdo,
                          centerAnalogR_Y - maxAnalogL_X,
                          maxAnalogR_Y - centerAnalogL_X,
                          maxPWM,
                          minPWM);

    pwmMotorEsquerdo = map(valorAnalogicoDireito + valorAnalogicoEsquerdo,
                           centerAnalogR_Y + centerAnalogL_X,
                           maxAnalogR_Y + maxAnalogL_X,
                           maxPWM,
                           minPWM);
  }
```

3.4.2. Frente/Esquerda

Se o analógico direito for pressionado para frente (+) e o analógico esquerdo for pressionado para a esquerda (-), o robô deve andar para frente e para esquerda. Para esse caso, a sentido de rotação de ambas as rodas deve ser igual. Entretanto, a velocidade da roda direita deve ser maior que a velocidade da roda esquerda, para que o robô vire para a esquerda.

```
else if (valorAnalogicoEsquerdo < (centerAnalogL_X - toleranciaAnalogico)) {

  pwmMotorDireito = map(valorAnalogicoDireito - valorAnalogicoEsquerdo,
                        centerAnalogR_Y - centerAnalogL_X,
                        maxAnalogR_Y - minAnalogL_X,
                        maxPWM,
                        minPWM);

  pwmMotorEsquerdo = map(valorAnalogicoDireito + valorAnalogicoEsquerdo,
                         centerAnalogR_Y + minAnalogL_X,
                         maxAnalogR_Y + centerAnalogL_X,
                         maxPWM,
                         minPWM);
}
```

3.4.3. Frente

Se o analógico direito for pressionado para frente (+) e o analógico esquerdo não for pressionado (nulo), o robô deve andar apenas para frente. Para esse caso, a velocidade e o sentido de rotação de ambas as rodas devem ser iguais;

```
  else {
    pwmMotorDireito = map(valorAnalogicoDireito, centerAnalogR_Y, maxAnalogR_Y, maxPWM, minPWM);
    pwmMotorEsquerdo = map(valorAnalogicoDireito, centerAnalogR_Y, maxAnalogR_Y, maxPWM, minPWM);
  }
}
```

3.4.4. Ré/Direita

Se o analógico direito for pressionado para trás (-) e o analógico esquerdo for pressionado para a direita (+), o robô deve andar para trás e para direita. Para esse caso, a sentido de rotação de ambas as rodas deve ser igual. Entretanto, a velocidade da roda esquerda deve ser maior que a velocidade da roda direita, para que o robô vire para a direita.

```
else if (valorAnalogicoDireito < (centerAnalogR_Y - toleranciaAnalogico)) {

  digitalWrite(SENTIDO_MOTOR_DIREITO, LOW);
  digitalWrite(SENTIDO_MOTOR_ESQUERDO, LOW);

  if (valorAnalogicoEsquerdo > (centerAnalogL_X + toleranciaAnalogico)) {
    pwmMotorDireito = map(valorAnalogicoEsquerdo + valorAnalogicoDireito,
                          centerAnalogL_X + minAnalogL_X,
                          maxAnalogL_X + centerAnalogR_Y,
                          minPWM,
                          maxPWM);

    pwmMotorEsquerdo = map(valorAnalogicoEsquerdo - valorAnalogicoDireito,
                           centerAnalogL_X - centerAnalogR_Y,
                           maxAnalogL_X - minAnalogR_Y,
                           minPWM,
                           maxPWM);
  }
```

3.4.5. Ré/Esquerda

Se o analógico direito for pressionado para trás (-) e o analógico esquerdo for pressionado para a esquerda (-), o robô deve andar para trás e para esquerda. Para esse caso, a sentido de rotação de ambas as rodas deve ser igual. Entretanto, a velocidade da roda direita deve ser maior que a velocidade da roda esquerda, para que o robô vire para a esquerda.

```
  else if (valorAnalogicoEsquerdo < (centerAnalogL - toleranciaAnalogico)) {
    pwmMotorDireito = map(valorAnalogicoEsquerdo + valorAnalogicoDireito,
                          centerAnalogL + centerAnalogR,
                          minAnalogL + minAnalogR,
                          minPWM,
                          maxPWM);

    pwmMotorEsquerdo = map(valorAnalogicoEsquerdo - valorAnalogicoDireito,
                           minAnalogL - centerAnalogR,
                           centerAnalogL - minAnalogR,
                           minPWM,
                           maxPWM);
  }
```

3.4.6. Ré

Se o analógico direito for pressionado para trás (-) e o analógico esquerdo não for pressionado (0), o robô deve andar para trás apenas. Para esse caso, a velocidade e o sentido de rotação de ambas as rodas devem ser iguais;

```
  else {
    pwmMotorDireito = map(valorAnalogicoDireito, centerAnalogR, minAnalogR, minPWM, maxPWM);
    pwmMotorEsquerdo = map(valorAnalogicoDireito, centerAnalogR, minAnalogR, minPWM, maxPWM);
  }
}
```

3.4.7. Direita no eixo

Se o analógico esquerdo for pressionado para a direita (+) e o analógico direito não for pressionado (0), o robô deve rotacionar no sentido horário. Para esse caso, a velocidade de rotação de ambas as rodas deve ser igual, entretanto, a roda da direita deve rodar para trás e a da esquerda, para frente (robô rotaciona no próprio eixo);

```
else {

  if (valorAnalogicoEsquerdo > (centerAnalogL_X + toleranciaAnalogico)) {
    digitalWrite(SENTIDO_MOTOR_DIREITO, LOW);
    digitalWrite(SENTIDO_MOTOR_ESQUERDO, HIGH);
    pwmMotorDireito = map(valorAnalogicoEsquerdo, centerAnalogL_X, maxAnalogL_X, minPWM, maxPWM);
    pwmMotorEsquerdo = map(valorAnalogicoEsquerdo, centerAnalogL_X, maxAnalogL_X, maxPWM, minPWM);
  }
```

3.4.8. Esquerda no eixo

Se o analógico esquerdo for pressionado para a esquerda (-) e o analógico direito não for pressionado (nulo), o robô deve rotacionar no sentido anti-horário. Para esse caso, a velocidade de rotação de ambas as rodas deve ser igual, entretanto, a roda da direita deve rodar para frente e a da esquerda, para trás (robô rotaciona no próprio eixo);

```
else if (valorAnalogicoEsquerdo < (centerAnalogL_X - toleranciaAnalogico)) {
  digitalWrite(SENTIDO_MOTOR_DIREITO, HIGH);
  digitalWrite(SENTIDO_MOTOR_ESQUERDO, LOW);
  pwmMotorDireito = map(valorAnalogicoEsquerdo, centerAnalogL_X, minAnalogL_X, maxPWM, minPWM);
  pwmMotorEsquerdo = map(valorAnalogicoEsquerdo, centerAnalogL_X, minAnalogL_X, minPWM, maxPWM);
}
```

3.4.9. Imobilidade

Se nenhum analógico for pressionado, o robô permanece imóvel.

```
  else {
    digitalWrite(SENTIDO_MOTOR_ESQUERDO, LOW);
    digitalWrite(SENTIDO_MOTOR_DIREITO, LOW);
    pwmMotorDireito = 0;
    pwmMotorEsquerdo = 0;
  }
}
```

Foi definida uma zona morta para o controle, ou seja, se os analógicos forem levemente pressionados, de forma que os valores lidos sejam menores do que a tolerância estipulada, o programa desconsidera o valor lido. Isso foi feito para evitar que o robô fique se movimentando sozinho, devido às vibrações mecânicas no controle e afins.

As variáveis centerAnalogR_Y e centerAnalogL_X representam, respectivamente, o valor que o analógico direito e esquerdo retornam quando não estão sendo pressionados.

As variáveis minAnalogR_Y e maxAnalogR_Y representam, respectivamente, o menor e o maior valor que o analógico direito pode retornar quando verticalmente movimentado (valor quando o joystick é inteiramente pressionado trás e para a frente).

As variáveis minAnalogL_X e maxAnalogL_X representam, respectivamente, o menor e o maior valor que o analógico esquerdo pode retornar quando horizontalmente movimentado (valor quando o joystick é inteiramente pressionado esquerda e para a direita).

Esses valores variam de controle para controle e podem ser obtidos conectando o controle à ESP32 com a utilização de um programa que será demonstrado mais a diante. Entretanto, é importante lembrar que, no código principal, o valor do analógico direito é multiplicado por -1 ao ser lido. Isso deve ser considerado na hora de definir os limites do controle.

O valor de velocidade colocado em cada roda é proporcional ao valor da combinação entre o valor horizontal do analógico esquerdo e o valor vertical do analógico direito (valorAnalogicoDireito ± valorAnalogicoEsquerdo) e vice-versa.

Conforme visto na apostila sobre Hardware, a saída analógica da ESP32 é capaz de gerar valores que vão de 0 a 255. Porém, a combinação dos valores dos analógicos pode gerar resultados que vão de -1024 a 1024 nos casos mais extremos. Portanto, é necessário converter o valor da combinação para um valor entre 0 e 255. Para tal tarefa, utilizamos o comando map(). Ele recebe como parâmetro o valor que se deseja converter, o limite inferior do intervalo de origem do valor, o limite superior do intervalo

de origem do valor, o limite inferior do intervalo de destino do valor e o limite superior do intervalo de destino do valor, nessa ordem. A partir dessas informações, ele calcula um novo valor dentro do intervalo de destino.

Para cada possível combinação dos analógicos (frente + direita, trás + esquerda, ...) existe uma faixa de valores para o intervalo de origem. Entretanto, o intervalo de saída será sempre de 0 a 255 ou de 255 a 0.

Para as condições em que os pinos de sentido são configurados como HIGH (analógico direito para frente), deve-se gerar valores de saída de 255 a 0 (0 quando a combinação dos joysticks atingir o valor máximo). Já para as condições em que os pinos de sentido são configurados como LOW (analógico direito para trás), deve-se gerar valores de saída de 0 a 255 (255 quando a combinação dos joysticks atingir o valor máximo). Isso se deve ao de funcionamento da ponte H. Caso queira se aprofundar mais na lógica de funcionamento da ponte H, consulte o manual (datasheet).

Finalizada a etapa de determinação, escreve-se os valores de velocidade nas saídas analógicas.

```
  Serial.print("PWM Direito: ");
  Serial.println(pwmMotorDireito);
  Serial.print("PWM Esquerdo: ");
  Serial.println(pwmMotorEsquerdo);

  analogWrite(VELOCIDADE_MOTOR_DIREITO, pwmMotorDireito);
  analogWrite(VELOCIDADE_MOTOR_ESQUERDO, pwmMotorEsquerdo);
}
```

Caso o robô seja/esteja desligado, os trechos de código contendo o mapeamento dos botões e dos analógicos e todo o processo de cálculo das velocidades será ignorado, a arma será desligada e a velocidade dos motores definida para 0.

```
else
  desligaRobo();
```

3.5. setup()

A função setup() é própria da ESP32 e, portanto, é **obrigatória** para o funcionamento do código. Ela é executada, de forma automática, uma única vez durante todo o funcionamento do programa: no momento em que o controlador é energizado.

De forma geral, neste trecho colocamos todas as configurações de funcionamento da ESP32 (definição dos pinos de entrada e saída, inicialização da serial, ...), inicializamos as bibliotecas utilizadas e definimos os estados iniciais das variáveis.

No código que estamos utilizando, primeiramente inicializamos a comunicação serial. É esta comunicação que utilizamos para exibir as mensagens de funcionamento no monitor serial. Em sequência, inicializamos a biblioteca Bluepad32 e removemos todos os dispositivos que haviam sido pareados anteriormente. Perceba que as funções de conexão e desconexão do controle são passadas como parâmetro durante a inicialização. Depois definimos os pinos nos quais a ponte H e o motor da arma estão conectados como saída (OUTPUT), utilizando o comando pinMode, e as desligamos (desligaRobo()).

```
void setup() {

  Serial.begin(115200);

  // Inicia comunicação Bluetooth que permite a conexão com controle
  BP32.setup(&onConnectedController, &onDisconnectedController);
  BP32.enableVirtualDevice(false);

  // Desparea os controles que haviam sido conectados anteriormente
  BP32.forgetBluetoothKeys();

  // Configura pinos da ESP32 como saída

  pinMode(SENTIDO_MOTOR_ESQUERDO, OUTPUT);
  pinMode(VELOCIDADE_MOTOR_ESQUERDO, OUTPUT);

  pinMode(SENTIDO_MOTOR_DIREITO, OUTPUT);
  pinMode(VELOCIDADE_MOTOR_DIREITO, OUTPUT);

  pinMode(PINO_ARMA, OUTPUT);

  // Desliga movimentação e arma do robô, por garantia.
  desligaRobo();
}
```

3.6. loop()

Assim que o setup() é executado, a ESP32 passa automaticamente a executar a função loop(). Ela é executada de forma interrupta, ou seja, o controlador executa tudo que está dentro dela, uma vez terminada a execução, volta para o início e executa novamente. Esse processo se repete até que a ESP32 seja desligada. Ela também é própria do controlador e, portanto, é obrigatória para o funcionamento do código.

A função loop() contida em nosso código é bastante reduzida, pois fizemos todo o descritivo do funcionamento do robô na função processControllers(). Neste trecho apenas verificamos se houve atualização nos dados do controle e, em caso afirmativo, executamos a função processControllers().

```
void loop() {
  bool dataUpdated = BP32.update();
  if (dataUpdated)
    processControllers();
}
```

4. Parametros.h

Na mesma pasta do código principal há um arquivo de configurações, denominado parametros.h (.header). Esse arquivo foi criado com o intuito de deixar o código mais organizado, facilitando a alteração de alguns parâmetros caso necessário.

Nas duas primeiras linhas temos o início do arquivo. Se ele não tiver sido definido (if not defined - #ifndef), o definimos (#define) como sendo parametros_h. Pode ser que tenhamos arquivos de configuração com o mesmo nome, por isso esses comandos se fazem necessários. Perceba que o nome na definição deve ser semelhante ao nome do arquivo, diferindo apenas no ponto/*underline*.

```
#ifndef parametros_h
#define parametros_h
```

Obs: Caso queira conhecer um pouco mais sobre essas diretivas, consulte esse link. (https://learn.microsoft.com/pt-br/cpp/preprocessor/preprocessor-directives?view=msvc-170)

Em sequência, incluímos a biblioteca Arduino.h, necessária para que possamos programar na IDE do Arduino e para que alguns comandos sejam compreendidos corretamente.

```
#include <Arduino.h>
```

Logo abaixo vem a definição dos pinos que estamos utilizando para controlar a arma e os motores de movimentação. Ao digitarmos no código o nome escrito após o comando #define estamos na verdade digitando o valor numérico que está do lado direito do nome. Essa definição é importante pois facilita a identificação dos pinos no código principal.

```
#define SENTIDO_MOTOR_ESQUERDO 33
#define VELOCIDADE_MOTOR_ESQUERDO 32

#define SENTIDO_MOTOR_DIREITO 26
#define VELOCIDADE_MOTOR_DIREITO 25

#define PINO_ARMA 14
```

Temos agora a definição de alguns parâmetros. Primeiramente vem os parâmetros dos analógicos do controle: valores centrais, máximos e mínimos. Em sequência, o valor da zona morta, ou seja, a partir de qual valor desejamos que o código passe a considerar a leitura do analógico como sendo diferente de 0, utilizado para evitar trepidações. Por último temos a definição dos limites da faixa de valores que será

utilizada para controlar a ponte H. É valido lembrar que estes valores estão relacionados à resolução da saída analógica (PWM) da ESP32.

A palavra const no início da definição dos parâmetros indica que eles não mudarão ao longo da execução do código, terão sempre o valor definido no arquivo .h. Sempre que uma dessas variáveis for escrita no código principal ela estará referenciando o valor descrito nesse arquivo.

```
// Comportamento natural do controle em ambos os analógicos
// Cima - Baixo +
// Direita + Esquerda -

const int32_t minAnalogR_Y = -508, centerAnalogR_Y = 0, maxAnalogR_Y = 512;  // Valores reais * -1
const int32_t minAnalogL_X = -512, centerAnalogL_X = 0, maxAnalogL_X = 508;

const int32_t toleranciaAnalogico = 20;  // zona morta do controle

// Pino de controle em HIGH: 255 (menor velocidade) a 0 (maior velocidade)
// Pino de controle em LOW: 0 (menor velocidade) a 255 (maior velocidade)

const int minPWM = 0, maxPWM = 255;
```

Na última linha do programa temos o comando #endif, encerrando a diretiva #ifndef que está no início do arquivo.

```
#ifndef parametros_h
#define parametros_h

#include <Arduino.h>

#define SENTIDO_MOTOR_ESQUERDO 33
#define VELOCIDADE_MOTOR_ESQUERDO 32

#define SENTIDO_MOTOR_DIREITO 26
#define VELOCIDADE_MOTOR_DIREITO 25

#define PINO_ARMA 14

// Comportamento natural do controle em ambos os analógicos
// Cima - Baixo +
// Direita + Esquerda -

const int32_t minAnalogR_Y = -508, centerAnalogR_Y = 0, maxAnalogR_Y = 512;  // Valores reais * -1
const int32_t minAnalogL_X = -512, centerAnalogL_X = 0, maxAnalogL_X = 508;

const int32_t toleranciaAnalogico = 20;  // zona morta do controle

// Pino de controle em HIGH: 255 (menor velocidade) a 0 (maior velocidade)
// Pino de controle em LOW: 0 (menor velocidade) a 255 (maior velocidade)

const int minPWM = 0, maxPWM = 255;

#endif
```

4.1. Determinação de parâmetros

Conforme visto anteriormente, para o correto cálculo dos valores de velocidade de cada roda do robô é necessário determinar a faixa de valores retornada por cada analógico do controle (vide explicação da função processControllers() no tópico sobre o código principal). Para tal tarefa iremos utilizar o programa descobrir_parametos_controle, disponível no repositório que você baixou no GitHub (https://github.com/nrc-cupim/cupim).

Ao abrir esse código, você irá perceber que ele é bastante semelhante ao código principal visto anteriormente, o diferencial está apenas nas funções processControllers() e onConnectedController().

Na função onConnectedController() foram adicionadas 4 novas linhas de código, que tem como principal objetivo fazer a leitura do endereço MAC do controle (em um tópico posterior irei explicar a funcionalidade desse endereço). Por enquanto, apenas anote o endereço que será exibido quando executarmos esse programa.

```
void onConnectedController(ControllerPtr ctl) {

  bool foundEmptySlot = false;

  for (int i = 0; i < BP32_MAX_GAMEPADS; i++) {
    if (myControllers[i] == nullptr) {
      Serial.println("\nAVISO: controle conectado.");
      myControllers[i] = ctl;
      foundEmptySlot = true;

      uint8_t addr[6];
      for (int i = 0; i < 6; i++)
        addr[i] = ctl->getProperties().btaddr[i];
      Serial.printf("Endereco MAC do controle: %x:%x:%x:%x:%x:%x\n\n", addr[0], addr[1], addr[2], addr[3], addr[4], addr[5]);
      break;
    }
  }

  if (!foundEmptySlot) {
    Serial.println("AVISO: Nao foi possivel conectar o controle.");
    Serial.println("AVISO: Reinicie a ESP32 e tente novamente.");
  }
}
```

Já a função processControllers() teve sua lógica substituída por um mapeamento de todas as entradas do controle. Ao executar o código e pressionar/mover os botões do controle, é possível ver como a ESP32 interpreta cada parâmetro.

```
void processControllers() {
  for (auto myController : myControllers) {
    if (myController && myController->isConnected() && myController->hasData()) {
      if (myController->isGamepad()) {
        Serial.printf(
          "idx=%d, dpad: 0x%02x, buttons: 0x%04x, axis L: %4d, %4d, axis R: %4d, %4d, brake: %4d, throttle: %4d, "
          "misc: 0x%02x, gyro x:%6d y:%6d z:%6d, accel x:%6d y:%6d z:%6d\n",
          myController->index(),        // Controller Index
          myController->dpad(),         // D-pad
          myController->buttons(),      // bitmask of pressed buttons
          myController->axisX(),        // (-511 - 512) left X Axis
          myController->axisY(),        // (-511 - 512) left Y axis
          myController->axisRX(),       // (-511 - 512) right X axis
          myController->axisRY(),       // (-511 - 512) right Y axis
          myController->brake(),        // (0 - 1023): brake button
          myController->throttle(),     // (0 - 1023): throttle (AKA gas) button
          myController->miscButtons(),  // bitmask of pressed "misc" buttons
          myController->gyroX(),        // Gyro X
          myController->gyroY(),        // Gyro Y
          myController->gyroZ(),        // Gyro Z
          myController->accelX(),       // Accelerometer X
          myController->accelY(),       // Accelerometer Y
          myController->accelZ()        // Accelerometer Z
        );
      }
    }
  }
}
```

```
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,    0, axis R:    0,    0, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,    0, axis R:    0,    0, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,    0, axis R:    0,    0, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,    0, axis R:    0,    0, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,  508, axis R:    0,    0, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,  508, axis R:    0,    0, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,  508, axis R: -204, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0001, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0001, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0001, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0000, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0001, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0001, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0001, axis L:    0,  508, axis R: -172, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0001, axis L:    0,  508, axis R: -132, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
idx=0, dpad: 0x00, buttons: 0x0001, axis L:    0,  508, axis R:    0, -512, brake:    0, throttle:    0, misc: 0x00, gyro x:    0 y:    0 z:    0, accel x:    0 y:    0 z:    0
```

Façamos agora um teste. Com o controle conectado, pressione a tecla *SELECT* e veja o que acontece com o valor da variável *misc*. Repita o processo para a tecla *START*. Por meio desses testes é possível entender as comparações (estadoMiscButtons == 0x02) e (estadoMiscButtons == 0x04) feitas no código principal.

```
uint16_t estadoMiscButtons = myController->miscButtons();

// Se SELECT for presionado, desliga robô.
if (estadoMiscButtons == 0x02) {
  roboLigado = false;
  Serial.println("Robo desligado");
}

// Se START for presionado, liga robô.
else if (estadoMiscButtons == 0x04) {
  roboLigado = true;
  Serial.println("Robo iniciado");
}
```

O mesmo raciocino se aplica à lógica de acionamento/desligamento da arma.

A partir do código de determinação dos parâmetros é possível alterar o funcionamento do robô (código principal) para a forma que achar melhor (configurar o acionamento e o desligamento do robô para os botões O e X, por exemplo). Para tal, basta fazer a leitura do parâmetro e comprar com o valor desejado. Por exemplo:

```
uint16_t estadoBotoes = myController->buttons();

// Se X for presionado, desliga robô.
if (estadoBotoes == 0x0001) {
  roboLigado = false;
  Serial.println("Robo desligado");
}

// Se O for presionado, liga robô.
else if (estadoBotoes == 0x0002) {
  roboLigado = true;
  Serial.println("Robo iniciado");
}
```

Esse mesmo código deve ser utilizado para fazer a determinação dos parâmetros centerAnalogR, centerAnalogL, minAnalogR, minAnalogL, maxAnalogR e maxAnalogL utilizados ao longo do código principal para definição do sentido e velocidade de rotação das rodas. Uma vez medidos, basta registra-los no arquivo parâmetros.h.

```
const int32_t minAnalogR_Y = -508, centerAnalogR_Y = 0, maxAnalogR_Y = 512;  // Valores reais * -1
const int32_t minAnalogL_X = -512, centerAnalogL_X = 0, maxAnalogL_X = 508;
```

Obs: Não esqueça que os valores do analógico direito devem ser multiplicados por -1 antes de serem colocados no arquivo (vide explicação do código principal)

É importante mencionar que controles diferentes podem ter comportamentos diferentes. Portanto, é recomendado testar cada controle antes de utiliza-lo para pilotar o robô.

5. Filtro MAC

Conforme mencionado anteriormente, cada controle possui um identificador, denominado endereço MAC. Este endereço é único para cada controle e serve para identificar dispositivos em uma conexão, permitindo que as mensagens sejam encaminhadas corretamente entre origem e destino.

Quando conectamos a ESP32 ao controle estamos na verdade estabelecendo uma conexão bluetooth entre esses dispositivos, permitindo que eles troquem dados entre si.

Pensando em um cenário de competição, é fácil perceber que poderão existir vários controles próximos ao robô. Portanto, é recomendado informar à ESP32 em qual controle ela deve se conectar, evitando que o sinal de outro controle interfira na comunicação e, por consequência, no funcionamento do robô.

Para implementar tal filtro, basta utilizar o código filtro_mac_controle, disponível na pasta que você baixo do GitHub (https://github.com/nrc-cupim/cupim).

Esse código habilita uma lista de permissões na ESP32 e adiciona endereços a ela. Dessa forma, apenas dispositivos adicionados à lista poderão se conectar ao controlador.

O endereço adicionado será armazenado na memória não volátil das ESP (NVS – Non volatile storage), assim como a configuração da lista de permissões. Isso significa que se o dispositivo reiniciar ou se outro código for carregado para ele, tanto o endereço quanto a configuração da lista de permissões ainda estarão salvos.

Para utilizar o código, o primeiro passo é escrever o endereço MAC do controle, obtido com o código descobrir_parametros_controle (também disponível no repositório do GitHub), no local informado.

```
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x0
mode:DIO, clock div:1
load:0x3fff0030,len:1344
load:0x40078000,len:13964
load:0x40080400,len:3600
entry 0x400805f0
Bluepad32 (C) 2016-2024 Ricardo Quesada and contributors.
Version: v4.0.2
BTstack: Copyright (C) 2017 BlueKitchen GmbH.
BTstack up and running at E4:65:B8:83:3B:7E
05 0C 09 01 A1 01 85 02 15 00 25 01 75 01 95 0A 09 EA 09 CD 09 B6 09 B

AVISO: controle conectado.
Endereco MAC do controle: 98:b6:a7:7e:3b:f9
```

```
#include <uni.h>  // inclusão da biblioteca que implementa o filtro

// Substitua o texto entre aspas pelo endereço MAC do seu controle.
static const char* endereco_mac_controle_1 = "98:b6:a7:7e:3b:f9";
```

Dentro da função setup(), o primeiro passo é converter o endereço MAC que acabamos de definir para uma forma de endereço compreensível pela ESP32. Para tal tarefa, utiliza-se a função sscanf_bd_addr(). Ela recebe como parâmetro o endereço MAC que se deseja converter e a variável onde o endereço convertido será salvo.

Feita a conversão, o próximo passo é adicionar o endereço à lista de permissão utilizando a função uni_bt_allowlist_add_addr(). Ela recebe como parâmetro o endereço MAC convertido que se deseja adicionar á lista. Caso deseje remover um

endereço adicionado anteriormente, basta utilizar o comando uni_bt_allowlist_remove_addr().

É possível armazenar até quatro endereços MAC na memória da ESP, permitindo a conexão de 4 controles distintos. Supondo que quatro endereços já estejam armazenados na memória é necessário remover um antes de adicionar outro. É importante comentar também que caso endereços duplicados sejam adicionados, nada acontece.

Por fim, a função uni_bt_allowlist_set_enabled() é a responsável por ativar/desativar a lista de permissões. Caso true seja passado como argumento para a ela, habilita-se a lista de permissões. Já false desabilita.

```
void setup() {

  // Converte endereço MAC e salva na variável endereco_1
  bd_addr_t endereco_1;
  sscanf_bd_addr(endereco_mac_controle_1, endereco_1);

  // Remove endereço bluetooth da lista de permissão
  // uni_bt_allowlist_remove_addr(endereco_1);

  // Adiciona endereço à lista de permissão.
  uni_bt_allowlist_add_addr(endereco_1);

  // Habilita lista de permissão.
  uni_bt_allowlist_set_enabled(true);

  // Desabilita lista de permissão.
  // uni_bt_allowlist_set_enabled(false);
}

void loop() {
  // Deve estar vazio.
}
```

Ao carregar o código para a ESP32, o filtro passa a valer. O próximo passo é carregar o código principal para o controlador e validar o funcionamento do filtro.

```
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:1344
load:0x40078000,len:13964
load:0x40080400,len:3600
entry 0x400805f0
Bluepad32 (C) 2016-2024 Ricardo Quesada and contributors.
Version: v4.0.2
BTstack: Copyright (C) 2017 BlueKitchen GmbH.
BTstack up and running at E4:65:B8:83:3B:7E
Ignoring device, not in allow-list: 41:42:37:53:4D:B2
Ignoring device, not in allow-list: 41:42:37:53:4D:B2
Ignoring device, not in allow-list: 41:42:37:53:4D:B2
```

Link do tutorial utilizado como base: https://bluepad32.readthedocs.io/en/latest/FAQ/#how-to-pair-just-one-controller-to-one-particular-board